Designing and Compiling Functional Java for the Fresh Breeze Architecture

by

William J. Jacobs

B.S. Computer Science and Engineering
Massachusetts Institute of Technology

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

[JUNE]
May 2009

Author_____
Department of Electrical Engineering and Compute Science
May 22, 2009

Certified by_____
Jack Dennis, Professor Emeritus
Thesis Supervisor

Accepted by_____
Arthur C. Smith
Professor of Electrical Engineering
Chairman, Department Committee on Graduate Theses

Designing and Compiling Functional Java for the Fresh Breeze Architecture

by

William J. Jacobs

Submitted to the
Department of Electrical Engineering and Computer Science

May 22, 2009

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The Fresh Breeze architecture is a novel approach to computing that aims to support a high degree of parallelism. Rather than striving for heroic complexity in order to support exceptional single-thread performance, as in the Pentium and PowerPC processors, it focuses on using a medium level of complexity with a view to enabling exceptional parallelized performance. The design makes certain sacrifices with regard to capability in order to achieve a lower degree of complexity. These design choices have significant implications for compiling for the architecture. In particular, Fresh Breeze uses immutable, fixed-size memory chunks rather than sequential, mutable memory to store data [1][2][3]. This work demonstrates Functional Java, a subset of the Java language that one can compile to Fresh Breeze machine code with relative ease, overcoming challenges pertaining to aliased arrays and objects. It also describes work on a compiler designed for the Fresh Breeze architecture and how the work overcomes various challenges in compiling Java bytecode to data flow graphs.

Thesis Supervisor: Jack Dennis
Title: Professor Emeritus

1. Introduction

The Fresh Breeze processor design is in many ways a departure from conventional design. It is designed to be highly parallelized, with simultaneous multithreading and hardware support for spawning method calls to other processors or cores.

The Fresh Breeze system makes use of a memory model that is considerably different from the standard model. In the standard memory model, memory is regarded as a linear array of mutable memory locations. In the Fresh Breeze memory model, memory is allocated to a thread of computation in 128-byte chunks, each of which is referenced by a unique UID or pointer. Each chunk may contain pointers to other chunks. Fresh Breeze memory chunks are immutable in the sense that a thread may create a chunk and write data into it, but the chunk becomes permanently fixed once shared with other threads. This behavior provides a guarantee that the heap will never contain cyclic structures, ensuring that the heap forms a directed acyclic graph. This enables efficient hardware garbage collection using reference counting [1][2][3].

The immutability of memory is well-suited to the functional programming paradigm, which is concerned with functions that produce output values appropriate to a given set of inputs, without altering the inputs to the function. In addition, immutable memory eliminates the concern of cache coherence by ensuring that any memory stored in a cache is always accurate. In order to expedite the allocation of memory, the Fresh Breeze system relies on hardware garbage collection that is implemented using reference counting. The usage of hardware-based memory allocation and deallocation greatly reduces the overhead typically associated with these operations. Furthermore, it solves problems associated with determining when a portion of memory that was available to multiple processors may be deallocated.

While the particular features of the Fresh Breeze model make the design well-suited to scientific and other computation-intensive applications, they pose interesting challenges for a compiler designed to target Fresh Breeze machine code. In particular, how can one ensure that the heap contains no reference cycles without unnecessarily burdening the programmer? The aim of this paper is to describe the work involved in overcoming these challenges. This project entails writing software that compiles Java bytecode to a data flow graph. In targeting the Java programming language, the intent was to enable the programmer to use a familiar language, while simultaneously enabling the compiler to produce efficient Fresh Breeze machine code. In addition, by using Java bytecode, we are able to leverage parsers and semantic analyzers that already exist for the Java programming language in building the Fresh Breeze compiler.

The compilation of a program into Fresh Breeze machine code is a five-step process, as illustrated in Figure 1. First, the Java compiler produces Java bytecode from Java code. This is converted into an intermediate representation, which is subsequently compiled into a data flow graph. Optimization converts a data flow graph into an optimized data flow graph. Finally, this data flow graph is compiled into Fresh Breeze machine code.

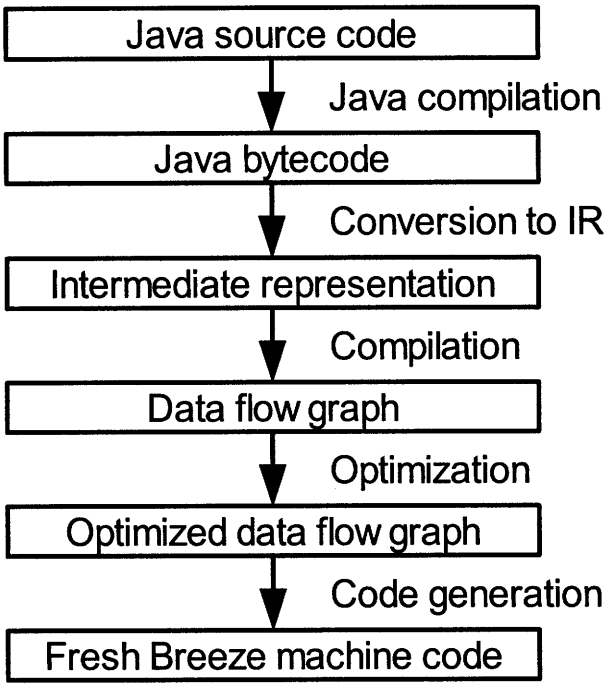

Figure 1: The Compilation Process

The compilation process will operate on a single Java class as the basic unit of compilation. Methods will be linked during the code generation phase by substituting the UID (pointer) of the appropriate code object at each method call.

This project focuses on the design and programming of the conversion of Java bytecode to an intermediate representation and subsequent compilation to data flow graphs. The objective is to produce functional data flow graphs from Java classfiles with minimal restrictions on the allowable Java source code. Generation of machine code for the Fresh Breeze architecture is left to future work.

The remainder of this thesis is organized into six sections. Section 2 describes a means of overcoming the problem of aliasing in order to design a compiler that is compatible with the Fresh Breeze memory model. Section 3 describes an intermediate representation to which Java bytecode is compiled and the algorithms used to compile to it. In Section 4, this paper describes

the data flow graph representation to which the intermediate representation is compiled as well as algorithms used in the compilation process. Section 5 details how the compiler checks restrictions on the Java language pertaining to aliasing. In section 6, this treats the subject of optimization, indicating how the compiler performs various optimizations. Finally, section 7 is concerned with how this work may be extended and improved.

2. Aliasing

2.1. Functional Java

In the Fresh Breeze memory model, memory is immutable. This makes operations such as changing an array's element or an object's field impossible, strictly speaking. Instead of changing an object or array, a program must produce a copy of the object or array that differs only in one of its fields or elements. This is consistent with the functional programming perspective, in which each method's purpose is to produce an output from a particular set of inputs without altering any of the inputs. However, it poses a unique challenge when compiling a non-functional language, such as Java.

In particular, aliasing presents a problem for a compiler designed to target Fresh Breeze machine code. Aliasing refers to the ability to access a given portion of memory using multiple symbolic names in the source code. This is problematic when a particular object is altered. Because the object might be accessible from multiple variables, the compiler would need to ensure that regardless of the variable from which the object is accessed, the contents are the same.

In order to use the Fresh Breeze memory model correctly, one can impose restrictions on the permissible Java programs that ensure that aliasing cannot occur. Having eliminated aliasing, each computation can be expressed functionally; that is, in such a manner that each method produces a set of outputs from a given set of inputs without modifying any of the inputs. This successfully eliminates the need for mutable memory. Furthermore, since the computation is functional, method calls are necessarily independent, allowing the compiler to maximize the amount of parallelism in a given program.

In practice, to express each Java method as a functional computation, one can design the compiler to implicitly return modified copies of all of its arguments. Figure 2 contains an example of a code listing and the equivalent Fresh Breeze pseudocode.

Code:

```
class Foo {
    int x;

    public static void bar(Foo f) {
        f.x = 1;
    }

    public static void baz(Foo f) {
        bar(f);
        f.x++;
    }
}
```

Fresh Breeze equivalent:

```
Foo bar(Foo f)
    Foo f2 = alteredCopy(f, x → 1)
    Return f2

Foo baz(Foo f)
    Foo f2 = bar(f)
    Foo f3 = alteredCopy(f2, x → (f2.x + 1))
    Return f3
```

Figure 2: Using Return Values to Implicitly Modify Arguments

This sort of compilation is capable of preserving Java semantics, so that the `bar` method does what the programmer would expect. However, aliasing presents a problem for this sort of approach. For example, consider the code and the Fresh Breeze equivalent presented in Figure 3.

Code:

```
class Foo {
    int x;
}

class Bar {
    Foo f;

    public static void baz(Bar b1, Bar b2) {
        b1.f.x = 1;
    }
}
```

Fresh Breeze Equivalent:

```
<Bar, Bar> baz(Bar b1, Bar b2)
    Foo f = b1.f
    Foo f2 = alteredCopy(f, x → 1)
    Bar b3 = alteredCopy(b1, f → f2)
    Return <b3, b2>
```

Figure 3: Problem Arising From Aliasing

A problem arises if `b1.f` and `b2.f` are the same. In this case, the programmer would expect both `b1.f.x` and `b2.f.x` to be changed to 1, in conformance with the standard Java semantics. However, if one were to use the code in Figure 3, only `b1.f.x` would be changed; `b2.f.x` would be unaltered.

There are two fundamental approaches to the problem of aliasing. In one approach, the program performs runtime checks to identify each instance of aliasing so that the compiler may take the appropriate action. For example, one could rewrite the above `baz` function as demonstrated in Figure 4:

```
<Foo, Foo> baz(Foo f1, Foo f2)
    Bar b = f1.bar
    Bar b2 = alteredCopy(b, x → 1)
    Foo f3 = alteredCopy(f1, bar → b2)
    If f2.bar = b
        f2 = alteredCopy(f2, bar → b2)
    Return <f3, f2>
```

Figure 4: Runtime Correction of Aliasing

This solution is undesirable, mainly because it requires an exceedingly large amount of

time in even some relatively common cases. For example, if a method is passed an array of 1,000 objects, every time one of the objects' fields is changed, the method would have to check each of the other objects to see if they match the one that was changed.

Another approach is to place restrictions on what Java programs the compiler will accept. This is the approach recommended here. By forbidding certain Java programs, one can ensure that the problem of aliasing is avoided.

Having settled on this approach, the objective is to place as little restriction on the programmer as possible. One might consider the simple solution of requiring objects to be immutable, rejecting any program that modifies any object's field, as proposed in [5]. However, a less restrictive approach is possible.

One may describe object references using a directed graph. Each node in the graph represents a single array or object. There is an edge from an array to each of its elements and from an object to each of its fields. The solution to the problem of aliasing described here is a set of compile-time checks designed to ensure that the graph of the references forms a forest. The checks also guarantee that the parent of every object or array that is referenced in a method is known at compile time.

The restricted subset of Java is termed "Functional Java". The restrictions are as follows:

1. No reference argument to a method call may be a descendant of another argument or the same as another argument. For example, the following is disallowed:

```
void foo() {
    Foo f = new Foo();
    bar(f, f.bar);
}
```

2. A method may not return one of its reference arguments or one of their descendants. This rule forbids code such as the following:

```
Bar foo(Foo f) {
    return f.bar;
}
```

3. No reference may be made a descendant of another reference, unless it is known to have no parent. This prohibits behavior such as the following:

```
void foo(Foo f, Bar b) {
    f.bar = b;
}
```

4. A reference variable may not be altered if the identity of its parent may depend on which branch is taken. The following, for example, is not permitted:

```
void foo(Foo f1, Foo f2) {
    Bar b;
    if (baz())
        b = f1.bar;
    else
        b = f2.bar;
    b.x = 1;
}
```

5. A variable containing a reference descendant of an array may not be accessed or altered after a descendant of the array is altered, unless the variable was set after the alteration was performed. This forbids the following code, which would be problematic if the inputs `a` and `b` were the same:

```
void foo(Foo[] f, int a, int b) {
    Foo f2 = f[a];
    Foo f3 = f[b];
    f2.x = 1;
    f3.x = 2;
}
```

6. A variable containing a reference descendant of an array may not be altered after the array is altered, unless the variable was set after the alteration was performed. The following is

forbidden by this rule, since a problem would arise if the inputs `a` and `b` were the same:

```
void foo(Foo[] f, int a, int b) {
    Foo f2 = f[a];
    f[b] = new Foo();
    f2.x = 2;
}
```

7. The == and != operators may not be used between any pair of references, unless at least one of them is explicitly **null**. At any rate, because of the above restrictions, the result of such an operation would be known at compile time (unless one of the operands is **null**), so the == or != was probably used in error.

Note that each of these conditions can be checked on a per-method basis. In particular, it is guaranteed that if two classes can successfully be compiled separately, then they can be successfully compiled together.

In addition, static fields are required to be compile-time constants, as proposed in [5]. Otherwise, every method would have to accept the static fields as implicit arguments and implicitly return the new values. At any rate, permitting variable static fields would not conform to the paradigm of functional programming.

2.2. Implementing the Java API Using Functional Java

Much of the Java API requires a violation of one of the aliasing rules. For example, the `ArrayList` class, which stores a dynamically sized array, has a method to add an element to the class. Figure 5 contains a simplified form of how it is implemented.

```
public void add(T element) {
    this.array[pos] = element;
    pos++;
}
```

Figure 5: Implementation of `ArrayList.add`

This code is forbidden by rule 3, since `element` is made a descendant of **`this`**, both of which are arguments to the method. In order to get around the restrictions, the programmer may use a Fresh Breeze library method of the form presented in Figure 6.

```
public class FBUtil {
    public static native <E> E copy(E object);
}
```

Figure 6: The `FBUtil.copy` *method*

The method takes an object or array and returns a deep copy of it. Counter-intuitively, the method takes no time at all. It simply returns its argument and causes the compiler to treat the returned object as if it were an unrelated object. This implementation is guaranteed to be correct because in the Fresh Breeze memory model, memory is immutable.

The `FBUtil.copy` method provides a workaround for the `ArrayList` method shown above. It may instead be implemented as presented in Figure 7:

```
public void add(T element) {
    this.array[pos] = FBUtil.copy(element);
    pos++;
}
```

Figure 7: An alternative implementation of `ArrayList.add`

This alters the semantics of the `ArrayList` class. Now, instead of storing references to objects, it would store copies of objects and would only permit retrieval of copies of its elements.

Another hitch pertaining to the standard Java library arises with regard to the `equals` function. For example, say a programmer were to implement a class `FBHashSet` that is similar to the Java `HashSet` class, except that it uses the copying semantics suggested above. Then, say the programmer uses the class as shown in Figure 8:

```
boolean foo() {
    Foo f = new Foo();
    FBHashSet<Foo> fs = new FBHashSet<Foo>();
    hs.add(f);
    return hs.contains(f);
}
```

Figure 8: Using `FBHashSet`

This method would not return **`true`** as expected, unless the `Foo` class correctly overrides the `Object.equals` and `Object.hashCode` methods. This is because the method adds a copy of `f` rather than a direct reference to `f` to the `FBHashSet`.

Some of the instance methods in the Java library are specified to return the objects on which they are called. Such methods are a violation of the second aliasing rule. One example of such a method is the one shown in Figure 9.

```
public class StringBuilder {
    public StringBuilder append(CharSequence s) {
        ...
        return this;
    }
}
```

Figure 9: Aliasing in the `StringBuilder` Class

Methods of the `StringBuilder` class are special from the perspective of the Java compiler, since expressions such as `str + "something"` are typically compiled by creating a temporary `StringBuilder` object and calling one of the `append` methods. In order to permit compiling such expressions, the compiler treats the return values of a hardcoded set of library methods that return **`this`** differently.

It is not immediately obvious how burdensome the restrictions imposed by Functional Java are on the programmer. To get a sense of this, and to enable users of the Fresh Breeze system to program using the standard Java API, it was useful to reimplement much of the Java API in order to satisfy these restrictions.

Experience suggests that for some tasks, it is difficult to conform to the restrictions. In particular, it is difficult to work with chains of objects that may be of arbitrary length, as in linked lists and binary search trees. The implementation of a red-black tree and a hash table uses integers to serve as pointers to nodes in the red-black tree and nodes in the linked lists in the hash table, which took difficulty to maintain correctly. However, for most tasks, it took little effort to keep from violating the aliasing restrictions, particularly with the assistance of a compiler that provided the file and line number of any such violations.

While reimplementing the Java API provides a useful barometer of the difficulty of

following the aliasing restrictions, the primary target of the Fresh Breeze system is computation-intensive scientific applications. For these applications, which often make less use of high-level constructs than other programs, it seems likely that following the aliasing rules would be easier. In general, while the restrictions may be limiting to people used to working with user-end PC's, who are often concerned with programs that require careful design and make use of Java's more advanced features, it is probably not very limiting to those writing scientific programs, who tend to rely on arrays and matrices for algorithms such as the Fast Fourier Transform and the Cholesky algorithm.

2.3. A Possible Extension to Functional Java

One possible extension to Functional Java would be to behave differently when dealing with immutable objects, which are understood to be objects consisting only of final, immutable fields whose constructors do nothing but set these fields. This would allow relaxation of the restrictions in certain cases. For example, one could allow immutable objects to be made children of multiple objects or arrays. Such an extension would cause Functional Java to be strictly less restrictive.

However, there is at least one reason why such an extension might not be desirable. Specifically, it is difficult to know the intent of the programmer when writing a class. A programmer might create a class that is initially very simple, which might by accident be immutable. The programmer might start using this class in several places with the intent of making the class mutable later on. He might inadvertently write code that would violate the aliasing restrictions if the class were mutable. When the programmer changes the class to be

mutable, numerous aliasing restrictions would appear.

At any rate, this possible extension affects the convenience of the language but not its power. Any case where one could refer to an immutable object with the extension but not without it can be corrected by a call to the `FBUtil.copy` method described in section 2.2.

2.4. Related Work

Previous work pertaining to aliasing has largely been focused on alias analysis within common languages such as C that permit aliasing. Such analysis is intended to improve optimization rather than eliminate aliasing [4]. This is understandable, since the memory model in use on conventional processors allows arbitrary mutation of data stored in memory.

Some functional languages allow the programmer to indicate occasions in which aliasing cannot occur. One way of doing so is uniqueness typing, which permits each object that the programmer declares to be linear to appear only once on the right-hind side of an assignment. Similarly, in a linear type system, once a value is cast to a linear form, the value can only be used once in the right-hand side of an assignment [7]. Like the work in this paper, such systems involve using compile-time checks to ensure that aliasing does not occur. However, these alternative systems are generally aimed at limiting aliasing rather than eliminating it outright. Furthermore, by comparison with the aliasing rules presented in section 2.1, they are particularly restrictive.

3. Compiling to Intermediate Representation

When compiling a program from Java bytecode to a data flow graph (DFG) representation, it is helpful to first convert the bytecode into an intermediate representation. The primary motivation for using this intermediate step is the fact that Java bytecode expresses some variables implicitly as positions on a stack, but it is more useful to refer to each variable explicitly. In addition, such a conversion enables the compiler to more easily keep track of the datatype of each variable that it is considering, which may enable certain optimizations at a later time.

The intermediate representation is based on the representation used in [5]. It consists of sequences of statements. Each particular type of statement is represented as a subclass of the abstract `IrStatement` class. Control flow is represented by `IrCondBranch` statements, which perform conditional branches, and `IrUncondBranch` statements, which perform unconditional branches. Statements typically operate on operands, which are represented as instances of `IrOperand`. Each `IrOperand` is either a fixed constant, using the `IrConstant` subclass, or a variable reference, using the `IrReference` subclass. `IrOperand` objects store the type of the operand. At this stage in compilation, the compiler does not determine whether a reference operand is an object or an array; this determination is deferred to a later time. The relevant class hierarchy appears in Figure 10.

```
IrStatement
      IrBranch
            IrCondBranch
            IrUncondBranch
IrOperand
      IrReference
      IrConstant
```

Figure 10: Partial Class Hierarchy for the Intermediate Representation

In order to convert the implicit information about variables on the stack and in the local variable array to explicit variable references, the compiler uses a `StackAndLocals` class. For each stack or local variable position, the `StackAndLocals` class keeps a map from each datatype to an `IrReference`. Each such `IrReference` is used to hold any value of its respective type that is stored in its respective position. In addition, the class keeps track of the datatype that is stored in each stack and local position at the current point in compilation. Using this construction, each push of a value X onto a stack or store operation of a value X into a local variable can be compiled as an assignment of the value X to the appropriate variable.

Using the `StackAndLocals` class, it is generally easy to compile bytecode to an intermediate representation. Java bytecodes represent relatively simple operations, such as pushing the value 2 onto the stack or converting the top item on the stack from an integer to a floating point value. For most bytecodes, the compiler can pop any operands it needs from the stack, perform the desired operation on them, store the result in a temporary variable, and push the variable onto the stack, using the `StackAndLocals` class to take care of the details of pushing and popping.

One challenge in producing the intermediate representation is determining the type of elements stored in each array on which an array get or set operation is performed. This type

includes the number of dimensions in the array and the type of elements stored in the array. Each array get or set operation is required to store the exact type of the array in question. Doing so gives the compiler the flexibility to store arrays and objects differently, to store multidimensional arrays as single-dimensional arrays if desired, to store arrays containing different types of elements differently, and to perform other optimizations.

It is not trivial to determine the type of each array because the aastore and aaload bytecodes, which set and get an array element respectively, only indicate that some sort of reference is being stored in or retrieved from an array. The compiler does not immediately know whether the reference is another array or an object. Furthermore, a given variable may be one type of array in one location in the bytecode and another type of array in another location. However, if the compiler observes the entire body of the method, there will necessarily be enough information to make the desired determinations.

Because the compiler may need to look at a method's entire body to determine array types, it defers the computation until it has compiled the whole method to its intermediate representation. Once a method is compiled, to compute the types of the arrays, the compiler relies on fixed-point computation. The aim of the computation is to determine the type of array stored in each array variable at the beginning of each basic block of the program. Here, a "basic block" is understood to be a maximal sequence of statements to which only the first can be jumped.

For each basic block, the algorithm first compute a sequence of "generated" mappings from variables to array types. It does this by maintaining a map from `IrReferences` to array types. It iterates through the statements in the basic block. If one can determine the type of

some array variable referred to in the statement, it adds an entry to the map, replacing any existing entry for the variable. For example, it would know the type of any array that is the return value of a method call or is a field it is retrieving from an object.

Once the algorithm has computed this information, it makes use of a fixed-point algorithm. The algorithm is a special case of the fixed-point algorithm that is commonly used. The algorithm appears in Figure 11.

```
Map<N, Map<K, V>> doFixedPointMapUnion(Map<N, Map<K, V>> initialValues,
Map<N, Map<K, V>> generated, Map<N, Set<K>> killed, Map<N, Set<N>> successors)
    Map<N, Set<V>> allValues
    For map ∈ initialValues
        For (key, value) ∈ map
            allValues[key] = allValues[key] ∪ {value}
    For map ∈ generated
        For (key, value) ∈ map
            allValues[key] = allValues[key] ∪ {value}

    Map<N, Set<Pair<K, V>>> initialValues2
    For (node, map) ∈ initialValues
        For (key, value) ∈ map
            initialValues2[node] = initialValues2[node] ∪ {(key, value)}

    Map<N, Set<Pair<K, V>>> generated2
    Map<N, Set<Pair<K, V>>> killed2
    For (node, map) ∈ generated
        For (key, value) ∈ map
            generated2[node] = generated2[node] ∪ {(key, value)}
            For value2 ∈ allValues[key]
                killed2[node] = killed2[node] ∪ {(key, value2)}

    For (node, keys) ∈ killed
        For key ∈ values
            For value ∈ allValues[key]
                killed2[node] = killed2[node] ∪ {(key, value)}

    Map<N, Set<K, V>> fixedPoint
    fixedPoint = doFixedPointUnion(initialValues2, generated2, killed2,
successors)

    Map<N, Map<K, V>> fixedPoint2
    For (node, pairs) ∈ fixedPoint
        Map<K, V> map
        Set<K> repeatedKeys
        For (key, value) ∈ fixedPoint
            If map[key] ≠ null
                repeatedKeys = repeatedKeys ∪ {key}
                map[key] = null
            Else If key ∉ repeatedKeys
                map[key] = value
        fixedPoint2[node] = map
    Return fixedPoint2
```

Figure 11: An Algorithm for Fixed-Point Computation With Maps

The method relies on the algorithm in Figure 12 as a subprocedure.

```
Map<N, Set<V>> doFixedPointUnion(Map<N, Set<V>> initialValues, Map<N, Set<V>>
generated, Map<N, Set<V>> killed, Map<N, Set<N>> successors)
    Map<N, Set<N>> predecessors
    For (node, nodes) ∈ successors
        For node2 ∈ nodes
            predecessors[node2] = predecessors[node2] ∪ {node}

    Map<N, Set<N>> in
    Map<N, Set<N>> out
    Set<N> changed
    For (node, succ) ∈ successors
        If initialValues[node] = null
            changed = changed ∪ {node}
            out[node] = generated[node]
        Else
            in[node] = initialValues[node]
            out[node] = in[node] ∪ generated[node] - killed[node]

    While changed ≠ {}
        N node = Any element in changed
        changed = changed - {node}
        Set<V> newIn
        For node ∈ predecessors[node]
            newIn = newIn ∪ out[node]

        If newIn ≠ in[node]
            in[node] = newIn
            out[node] = newIn ∪ generated[node] - killed[node]
            For node2 ∈ successors[node]
                changed = changed ∪ {node2}
    Return in
```

Figure 12: Fixed-Point Algorithm

The method `doFixedPointUnion` is a standard and well-studied algorithm. In practice, implementations of the algorithm operate by storing sets of values as bit vectors, which enables them to perform the union and difference operations efficiently.

The idea of the `doFixedPointMapUnion` algorithm is to make use of the standard fixed-point algorithm by representing mappings as sets of key-value pairs. The algorithm converts the generated and killed mappings into generated and killed key-value pairs, passes these pairs to the ordinary fixed-point algorithm, and converts the results into the appropriate

maps. Note that in the case of determining array types, there are no killed keys, while the `initialValues` map consists of a mapping from the first basic block to the types of each array argument to the method.

In fact, this algorithm by itself is insufficient to determine every array type. Sometimes, it is necessary to know one variable's array type before another variable's array type can be discovered. In reality, the compiler runs the algorithm several times, each time using information from the previous run to expand the set of generated mappings, until the algorithm produces the same results twice in a row. Once the compiler has determined the type of each relevant array variable at the beginning of each basic block, it is a relatively simple matter to determine the type of each relevant array variable within each basic block.

Once the program has compiled each method to its intermediate representation, it inline the "access" methods. Access methods are methods the compiler generates that are designed to assist in information hiding. Access methods are of three types: retrieving a private field, setting a private field, and calling a private method. Nested classes that have access to these private fields and methods use such access methods rather than directly accessing the fields and methods. Inlining these methods enables the compiler to correctly identify violations of the aliasing rules presented in section 2.1 when compiling the intermediate representation to data flow graphs.

After producing a sequence of statements for the intermediate representation, the compiler creates a corresponding control flow graph. The control flow graph is essentially the same as the sequence of statements, except that it includes nodes to represent the basic blocks and edges to represent conditional or unconditional jumps. Creating the control flow graph

makes it easier to compile the intermediate representation into a data flow graph.

4. Compiling to Data Flow Graphs

4.1. Introduction to Data Flow Graphs

The compiler represents each method in a program as a data flow graph, using a representation inspired by [5] and [6]. Data flow graphs are contrasted with control flow graphs (CFGs), which are a more common representation. Control flow graphs represent a program as a graph of basic blocks, each consisting of a sequence of statements. Each node has edges to the nodes to which it might subsequently branch. By comparison, a data flow graph represents a program as a directed, acyclic graph containing nodes representing various operations. Each node's inputs are indicated by the incoming edges.

Data flow graphs are particularly appropriate for functional programming. Because no statement has any side effects, the predecessor-successor relationships represented in a data flow graph are a natural representation of the order in which computations must be performed. A data flow graph represents control flow implicitly using special constructs like conditional graphs.

To get a sense for how data flow graphs can represent a method, some examples ought to be of assistance. Figure 13 provides a data flow graph representation of the absolute value method.

Code:

```
int abs(int x) {
    if (0 <= x)
        return x;
    else
        return -x;
}
```

Data Flow Graph:

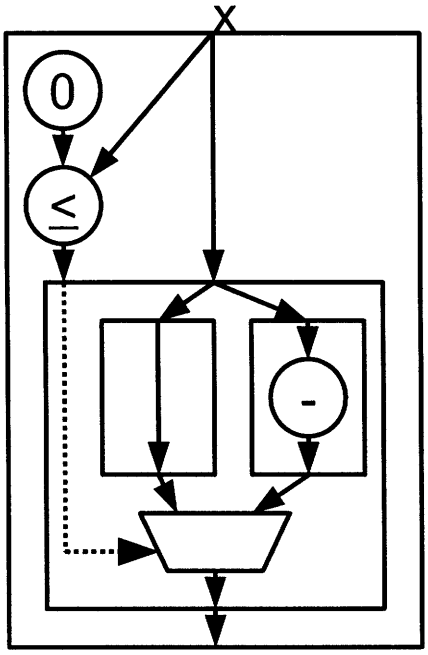

Figure 13: Data Flow Graph Representation of the Absolute Value Function

The data flow graph for the absolute value method begins by comparing the input to 0. The result of this computation is provided as an input to a conditional graph, which is a subgraph of the method. Depending on the result of the comparison, the conditional graph executes either its left subgraph or its right subgraph. In the former case, it returns the input to the method; in

the latter case, it negates the input and returns the result.

As another example, consider the data flow graph in Figure 14 for the method `foo`, which creates a new array and initializes it to all ones.

Code:

```
int[] foo(int size) {
    int[] array = new int[size];
    for(int i = 0; i < size; i++)
        array[i] = 1;
    return array;
}
```

Data Flow Graph:

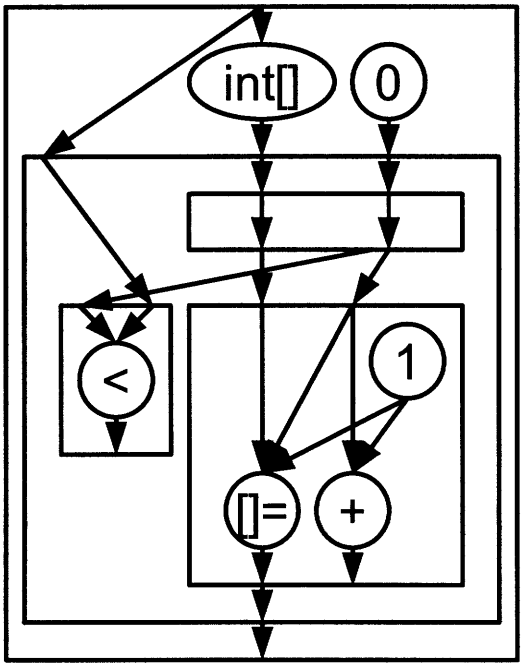

Figure 14: Data Flow Graph Representation of Building a New Array

In this example, the input is passed to a node that creates a new integer array of that length. Then, the result is passed to a loop graph. The loop graph consists of an initialization subgraph, a test subgraph, and a body subgraph. The initialization subgraph, which is the one on top, computes the initial values for the loop-dependent variables, `array` and `i` in this case. The test subgraph, which is on the left, checks whether `i` is less than `size`; if so, the method perform another iteration of the loop. The body of the loop, which is on the bottom-right, takes the values stored in the loop-dependent variables `array` and `i` at the beginning of one iteration and computes the values that are stored in these variables at the end of the iteration. The addition node in the body increments the loop counter. The node labeled []= takes as inputs an array, an index, and a value and outputs an array that is the same as the input array, but with the element of the given index set to the given value. The final value of `array` is used for the return value of the method.

Each data flow graph and subgraph contains a source node and a sink node. A source node accepts no inputs and outputs the inputs to the corresponding graph. Likewise, a sink node has no outputs, and its inputs are used as the outputs to the corresponding graph. To make data flow graphs easier to visualize, the source nodes and sink nodes are not shown in the diagrams for DFGs.

Although producing a correct data flow representation of a program is relatively difficult, the usage of data flow graphs generally simplifies the process of transforming and optimizing a given program. In control flow graphs, one is often concerned with keeping track of what is stored in each variable at any given time. Data flow graphs, by contrast, eliminate the need for such bookkeeping by abstracting the locations of intermediate results of the computation.

Determining where to store each of the intermediate values is left to the code generator, which is responsible for converting the data flow graphs to machine code.

4.2. Identifying Loops

The process of compiling the intermediate representation to its data flow graph representation is a relatively difficult one. To begin with, the compiler converts the sequence of statements in the intermediate representation into a control flow graph. Then, it order the nodes in the control flow graph into an array so that each loop occupies a contiguous range of elements in the array. Doing so makes it easier when it eventually compiles the loops.

As it turns out, it is not trivial to identify the loops in a particular program. A simple definition of a loop is "a maximal set of nodes for which there is a path from each node to every other node." However, this definition is insufficient for the purposes of the compiler. For example, consider the loop shown in Figure 15.

```
for(int i = 0; i < n; i++) {
    for(int j = 0; j < n; j++) {
        if (i + j == 5)
            return j;
    }
}
```

Figure 15: Nested Loops

Using the simple definition of a loop, the node containing the statement `return j` is not contained in the inner loop. In this case, since the statement can be reached from the inner loop, one would say that it is contained in the inner loop. This suggests a refined definition of a loop:

"A loop is a maximal set of nodes S for which there is a path from each node to every other node, along with the set of nodes T that cannot exit the loop without reaching a return statement."

In fact, even this definition is insufficient for the purposes of the compiler. Consider the code shown in Figure 16.

```
int i;
for(int i = 0; i < n; i++) {
    if (i == 5) {
        i = 3;
        break;
    }
}
i += 2;
```

Figure 16: Interruption of Control Flow in a Loop

Here, the statement `i = 3` ought to be contained in the first loop, but according to the above definition, it is not in any loop. Cases like this pose an additional problem for compilation.

To address these problems, when the compiler orders the statements, it first orders them so that all loops occupy a contiguous range of nodes, if one excludes "dead end" nodes and "break" nodes. "Dead end" nodes are defined to be nodes that cannot exit the loops in which they appear without reaching a return statement, while "break" nodes are defined to be nodes that can be included in a loop but cannot reach the beginning of the loop. Then, the compiler "fixes" the dead end and break nodes so that they are in the correct positions.

The `orderBeforeFix` method, in which the compiler establishes a partially correct ordering, appears in the appendix. The algorithm goes through the nodes in a particular order, so that the first node in each nested loop is visited before any other node in the nested loop and each

node that is not in a nested loop is visited before any of its successors. For each node, it determines whether the node is in a loop. If so, it computes the nodes in that loop, stores them in a group, and adds the group to a set of groups. Otherwise, it stores only the node itself in a group and adds that group to the set. After grouping the nodes, it orders the groups topologically, putting each group before all of its successors. Finally, it produces an ordering of nodes by going through the groups in order, recursively ordering each group and adding the nodes in order to the output list.

To identify the nodes contained in a loop, the compiler first locates every subsequent node that can reach the first node in the loop using dynamic programming. It checks each node that is a successor of some loop node to see whether it is a dead end. It adds such dead end nodes to the `deadEnds` set, and it adds each such node along with every node it can reach to the set of loop nodes. Then, the compiler remove each node with a predecessor that is not in the loop from the set of loop nodes.

Having roughly ordered the nodes, the compiler needs to fix the locations of break nodes and dead end nodes. It begins by computing the successor of each loop; that is, the node which is executed immediately after each loop in "normal" control flow. It uses the `computeLoopSuccessors` algorithm presented in the appendix.

Computing the successor of each loop is relatively complicated because there are cases in which one is constrained to make a dead end node the successor of a loop. For example, consider the code in Figure 17:

```
Outer: for(int i = 0; i < n; i++) {
Inner: for(int j = 0; j < n; j++) {
         for(int k = 0; k < n; k++) {
             if (i + j + k == 6)
                 break Inner;
         }
         if (i + j == 4)
             continue Outer;
     }
     return 3;
}
```

Figure 17: Interruption of Control Flow in Nested Loops

In this case, the compiler needs to ensure that the successor of the second for loop is the statement **return** 3 rather than the statement i++. More generally, it needs to ensure that dead end nodes with predecessors in multiple loops are the successor of some loop.

The algorithm operates by identifying all dead end nodes that have predecessors in multiple loops. For each, it identifies the children of the lowest common ancestor of the loops that contain the predecessors of the dead end node. For instance, in the above code, the statement **return** 3 is a dead end node that has predecessors in multiple loops, namely the inner two loops. The lowest common ancestor of these loops is the second loop, and the only child of this is the innermost loop. For each loop, the algorithm also identifies the set of nodes within the loop have a predecessor in an inner loop.

Next, the compiler orders the loops so that each loop appears before the loop in which it is enclosed, and it goes through the loops in order. The current loop will have a set of nodes X containing zero, one, or two nodes that have a predecessor in an inner loop, if one excludes break nodes. Each node in X will be either the successor of the loop or the node to which a continue statement from an inner loop jumps. There is also a set of nodes Y consisting of dead end nodes

that need to be the successor of the loop or one of its ancestors. If X has no nodes, take a node from Y and make it the successor of the loop. If X has two nodes, make the earlier of the two the successor of the loop. If X has one node, the algorithm needs to determine whether the node may be the target of a continue statement from an inner loop. This is the case if every path from the start of the loop back to the start passes through the node. If the node may be the target of a continue statement, make an element of Y the successor of the loop; otherwise, make the node in X the successor.

Having computed the successor of each loop, the algorithm can proceed to fix the ordering of the nodes. The `fixOrder` method and the `order` method that makes use of it appear in the appendix.

4.3. Identifying If Statements

Another challenge related to the inability to directly express control flow statements in data flow graphs pertains to if statements. In the absence of explicit branching, the compiler expresses conditional statements in data flow graphs as `DfgConditionalGraphs`. Each `DfgConditionalGraph` accepts a special boolean input edge. If the input evaluates to true, the program executes the left subgraph of the `DfgConditionalGraph`; otherwise, it executes the right subgraph. The outputs of the subgraphs flow into `DfgMergeNodes`, each of which has one input per subgraph and one output. Each `DfgMergeNode` selects as its output the input that corresponds to the subgraph that was executed.

In order to be able to compile if statements, the first goal is to locate the beginning and end of each if statement. The beginning of an if statement is understood to refer to the first CFG

node containing a conditional branch for the if statement, while the end of an if statement refers to the node where a program continues execution after executing the if statement. To identify the beginning nodes, the compiler goes through the nodes in order, so that it visits each node before any of its successors (ignoring the edges to the beginning of each loop). Whenever it finds the beginning of an if statement, it computes its end and the beginning nodes of each of its branches. Then, it recursively identifies if statements that are contained in each of the if statement's branches. The algorithm continues searching for the beginnings of if statements at the ending node of the if statement.

To compute the end of each if statement given its beginning, one may use the algorithm in Figure 18:

```
void findReachableNodes(Node start, Node end, Loop loop, Set<Node>
&reachable, Map<Node, Loop> enclosingLoops, Boolean isFirstCall)
    If start ∈ reachable
        Return
    reachable = reachable ∪ {start}
    If start = end
        Return
    Node enclosingLoop = enclosingLoops[start]
    Node loopStart
    If enclosingLoop ≠ null
        loopStart = enclosingLoop.start
    Else
        loopStart = null

    If (start = loopStart and Not isFirstCall) or enclosingLoop ≠ loop
        If enclosingLoop.successor ≠ null
            findReachableNodes(enclosingLoop.successor, end, loop, reachable,
enclosingLoops, false)
    Else
        For succ in start.successors
            If succ ≠ loopStart
                Loop loop2 = enclosingLoops[succ]
                If loop2 = loop or (loop2 ≠ null and loop2.parent = loop)
                    findReachableNodes(succ, end, loop, reachable,
enclosingLoops, false)
```

```
Node findIfStatementEnd(Node start, Node except, Map<Node, Loop>
enclosingLoops, Map<Node, Set<Loop>> loopSuccessors, Map<Node, Int>
nodeIndices)
    Set<Node> reachable
    findReachableNodes(start, except, enclosingLoops[start], reachable,
enclosingLoops, true)
    reachable = reachable - {except}

    Loop enclosingLoop = enclosingLoops[start]
    Node loopStart
    If enclosingLoop ≠ null
        loopStart = enclosingLoop.start
    Else
        loopStart = null

    Map<Node, Int> unreachablePreds
    unreachablePreds[start] = 0
    PriorityQueue<Node> queue sorted by predicate(node1, node2)
        If node1 ≠ loopStart and node2 = loopStart
            Return node1
        Else If node1 = loopStart and node2 ≠ loopStart
            Return node2
        Else If unreachablePreds[node1] = 0 and unreachablePreds[node2] > 0
            Return node1
        Else If unreachablePreds[node1] > 0 and unreachablePreds[node2] = 0
            Return node2
        Else If nodeIndices[node1] < nodeIndices[node2]
            Return node1
        Else If nodeIndices[node1] > nodeIndices[node2]
            Return node2
        Else
            Return null
    queue = {start}
    Set<Node> visited
    Boolean first = true
    While |queue| > 1 or first
        first = false
        Node node = First element in queue
        queue = queue - {node}
        visited = visited ∪ {node}
        Set<Node> ss
        If enclosingLoops[node] = enclosingLoop
            ss = node.successors
        Else
            ss = {enclosingLoops[node].successor}
        For succ ∈ ss
            If succ = except
                Continue
            If enclosingLoops[succ] = enclosingLoop or (enclosingLoops[succ]
≠ null and enclosingLoops[succ].parent = enclosingLoop)
                Int count = unreachedPreds[succ]
                If count = null
                    //Compute the number of unreachable predecessors
                    Set<Node> preds = succ.predecessors
```

```
                    For loop ∈ loopSuccessors[succ]
                        If loop.parent = enclosingLoop
                            preds = preds ⋃ {loop.start}
                    count = 0
                    For pred ∈ preds
                        If pred ∈ reachable and pred ∉ visited and pred ≠
succ
                            count = count + 1
                    unreachablePreds[succ] = count
                    queue = queue ⋃ {succ}
                Else
                    unreachablePreds[succ] = count - 1
    If |queue| > 0
        Node node = First element in queue
        If (enclosingLoops[node] = enclosingLoop or (enclosingLoops[node] ≠
null and enclosingLoops[node].parent = enclosingLoop)) and (enclosingLoop =
null or node ≠ loopStart)
            Return node
        Else
            Return null
    Else
        Return null


Node findIfStatementEnd(Node start, Map<Node, Loop> enclosingLoops, Map<Node,
Set<Loop>> loopSuccessors, Map<Node, Int> nodeIndices)
    Return findIfStatementEnd(start, null, enclosingLoops, loopSuccessors,
nodeIndices)
```

Figure 18: Finding the End of an If Statement

Note that in the pseudocode representation of this algorithm, as in all pseudocode presented in this paper, an ampersand in front of a parameter indicates that changes to the parameter are reflected in the calling method; otherwise, changes are not reflected in the caller.

The general approach of the algorithm is to maintain a priority queue of nodes. It sorts the nodes by whether each of their predecessors that is reachable from the starting node has been visited yet, then by their indices in the previously established node ordering. At each iteration, it extracts an element from the queue and add its successors to the queue. It continues until the queue has one or zero elements left. The remaining element, if any, is typically the end of the if

statement.

Once the compiler has located the end of an if statement, it identifies its branches. Each if-else statement will have two branches, which begin at two particular nodes, while each if statement will have one branch, beginning at a particular node. For the sake of consistency, the compiler considers if statements without else branches to have two branches: one for the contents of the if statement and one at the end of the if statement.

Often, the branches of the if statement are simply the successors of the beginning node of the if statement. However, the ternary `? :` operator and the short circuiting behavior of the `&&` and `||` operators pose a particular difficulty in identifying the branches, because in general, an if statement's condition may consist of multiple nodes. Intuitively, the branches are the earliest disjoint groups of nodes beginning with "choke points" in the graph through which a program must pass to go from the beginning of the if statement to the end. The example in Figure 19 illustrates one occasion when it is difficult to programatically identify the branches of an if statement.

Code:

```
if ((a ? b : c) || (d ? e : f))
    x = 1;
else
    return 2;
y = 3;
```

Data Flow Graph:

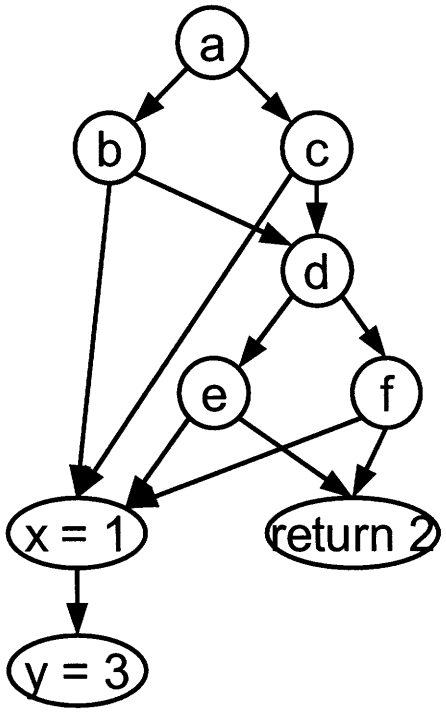

Figure 19: Data Flow Graph Representation of a Complex Condition

To compute the branches of an if statement, the compiler uses the `findBranches` algorithm shown in Figure 20.

```
void findBreakNodes(Node start, Node end, Set<Node> visited, Set<Node>
&breakNodes, Map<Node, Loop> enclosingLoops, Map<Node, Loop> loopStarts)
    If start = end or start ∈ visited
        Return
    visited = visited ∪ {start}
    For succ ∈ start.successors
        If succ = end
            Continue
        If succ.successors = {}
            breakNodes = breakNodes ∪ {succ}
        Else If enclosingLoops[succ] = enlcosingLoops[start]
            findBreakNodes(succ, end, visited, breakNodes, enclosingLoops,
loopStarts)
        Else
            Loop loop = loopStarts[succ]
```

```
            If loop ≠ null and loop.successor ≠ null and loop.parent =
enclosingLoops[start]
                findBreakNodes(loop.successor, end, visited, breakNodes,
enclosingLoops, loopStarts)
                breakNodes = breakNodes ⋃ {succ}


Node latestNodeBeforeBranches(Node start, Node end, Set<Node> &breakNodes,
Map<Node, Int> nodeIndices, Map<Node, Loop> enclosingLoops)
    Set<Node> reachable
    findReachableNodes(start, end, enclosingLoops[start], reachable,
enclosingLoops, true)
    PriorityQueue<Node> reachableSorted sorted by predicate(node1, node2)
        If nodeIndices[node1] < nodeIndices[node2]
            Return node1
        Else If nodeIndices[node1] > nodeIndices[node2]
            Return node2
        Else
            Return null
    reachableSorted = reachable
    Map<Node, Set<Node>> reachable2
    For node ∈ reachableSorted
        If node.nextIfFalse ≠ null
            List<Set<Node>> reachable3
            For succ ∈ node.successors
                If succ ∈ reachable and succ ∉ breakNodes
                    Set<Node> reachable4 = reachable2[succ]
                    If reachable4 = null
                        reachable4 = {}
                        findReachableNodes(next, end, enclosingLoops[node],
reachable4, enclosingLoops, true)
                        reachable2[succ] = reachable4
                    Add reachable4 to reachable3
                Else
                    Add {succ} to reachable3
            If (reachable[0] ∩ reachable[1]) - {except} = {}
                Return node
    Return start


Pair<Node> findBranches(Node start, Node end, Map<Node, Loop> enclosingLoops,
Map<Node, Int> nodeIndices, Map<Node, Loop> loopStarts)
    Set<Node> breakNodes
    findBreakNodes(start, end, {}, breakNodes, enclosingLoops, loopStarts)
    Node latest = latestNodeBeforeBranches(start, end, breakNodes,
nodeIndices, enclosingLoops)
    Node enlosingLoop = enclosingLoop[start]
    If enclosingLoops[latest] ≠ enclosingLoop
        Return (latest, null)
    Else If end ≠ null and ((latest.next = end and latest.nextIfFalse ∉
breakNodes) or (latest.nextIfFalse = end and latest.next ∉ breakNodes))
        Return (end, findIfStatementEnd(start, end))
    Else
        Node branch1 = latest.next
```

```
        If enclosingLoop ≠ null and enclosingLoop.start = branch1
            branch1 = null
        Node branch2 = latest.nextIfFalse
        If enclosingLoop ≠ null and enclosingLoop.start = branch2
            branch2 = null
        If branch1 = null
            branch1 = branch2
            branch2 = null
        Return (branch1, branch2)


IfStatementInfo computeIfStatement(Node start)
    Node end = findIfStatementEnd(start)
    Pair<Node> branches = findBranches(start, end)
    If branches.second = null
        end = null
    IfStatementInfo info
    info.start = start
    info.end = end
    info.branch1 = branches.first
    info.branch2 = branches.second
    Return info
```

Figure 20: Pseudocode For `findBranches`

The general strategy of this algorithm is to identify the latest node that is before either of the branches. Such a node is usually the earliest node with two successors for which the set of nodes reachable from each of its successors is disjoint. In addition to computing the branches of an if statement, the `computeIfStatement` method also corrects for certain cases in which the end of the if statement was initially computed incorrectly.

4.4. Compiling Data Flow Graphs

To compile a method to a data flow graph, the compiler make use the `Frame` class. The intent of the class is to keep track of the edges that contain the values of each of the variables. The `Frame` class operates in a manner analogous to a scope in programming languages. Each

`Frame` stores a reference to a parent `Frame`, which may be null. Whenever the compiler needs to obtain the value of a variable in a frame, it sees whether the frame has an edge for that variable. If so, it uses that edge. If not, it recursively obtains the desired edge from the parent frame.

The usage of frames enable the compiler to separately keep track of the location of each variable in a given context. For instance, consider the Java program in Figure 21.

```
x = 1;
if (condition)
    x = 2;
else
    y = 3 * x;
```

Figure 21: A Java Method Illustrating the `Frame` Class

When compiling the else branch of the if statement, the compiler need to determine the edge to which `x` corresponds. A naïve approach would use a global mapping from variable to edge. However, in this case, such a map would indicate to use the edge containing the value 2. If one uses frames instead, one will create new frames for each branch of the if statement, giving the correct result.

The `Frame` class stores additional mappings used to ensure that compilation properly implements the Fresh Breeze memory model. Recall that in order to meet the requirements of the memory model, programs do not modify objects and arrays in place; rather, they create altered copies of the objects and arrays and subsequently refer to these modified copies. Consider, for example, code in Figure 22.

```
Foo x = new Foo();
x.bar();
Foo y = x;
return y.baz();
```

Figure 22: A Java Method Illustrating the `ClassReference` *class*

In the last line of the code, the compiler needs to ensure that it calls `baz` on correct copy of `x`. To ensure correct behavior, the `Frame` class stores a mapping from each array or object variable to a `ClassReference` describing which object is stored in the variable. The `Frame` class also stores a mapping from each `ClassReference` to the edge that holds the current value of the array or object. In the above example, the compiler would create a `ClassReference` object `c` describing the `Foo` object that is constructed, and it would add a mapping indicating that `x` stores the value `c`. The statement `x.bar()` would create a method call node with an output indicating the altered copy of `x`. The complier would update the mapping for the edge for `c` to be this output edge. In the statement `Foo y = x`, it would add a mapping from the variable `y` to the `ClassReference c`. When compiling the last statement, it would observe that `y` contains the value `c`, so it would look up the edge that contains that value and pass it as an input to the `baz` method.

Furthermore, each `Frame` needs to keep track of the `ClassReference` parent of each `ClassReference` as well as the object field or array index from which it was derived. Every time an object or array is changed, the appropriate changes need to be affected in the ancestors of the object. These changes are manifested by creating `DfgArraySetNodes` and `DfgSetFieldNodes` for the ancestors and updating the locations of the ancestor

`ClassReferences` to the outputs of these nodes.

Doing this entails causes certain inefficiencies in the resulting data flow graph. Consider, for example, the Java method in Figure 23.

```
int[] foo(int[][] array) {
    int[] array2 = array[0];
    array2[0] = 1;
    array2[1] = 3;
    array2[2] = 7;
}
```

Figure 23: A Java Method Illustrating Unnecessary Array Setting

Each of the last three statements results in two array set operations: one that alters `array2` and one that alters `array`. However, it is possible to execute the entire method with only four array set operations, three of which alter `array2` and one of which alters `array`. For the time being, the compiler allows for such inefficiencies; correcting them is relegated to the optimization phase of compilation.

The process of compiling a control flow graph into a data flow graph is recursive. Having computed where a method's loops and if statements are located, the compiler proceeds with a call to `compileRange`, which compiles a sequence of nodes, if statements, and loops to a data flow graph. Each individual node is compiled by calling `compileNode`, while if statements and loops are compiled using calls to `compileIfStatement` and `compileLoop` respectively. `compileIfStatement` and `compileLoop` will use `compileRange` as a subprocedure to compile the branches of if statements and the bodies of loops.

Compiling a single node in the control flow graph is relatively straightforward. It entails

going through the statements one-by-one and producing DFG equivalents for each. An addition statement, for example, will be compiled to an addition node that accepts the operands as inputs, while a statement that sets an array element is compiled to a node whose inputs are the array, the index of the element the program is setting, and the value to which it is setting the element. When compiling a statement, the compiler use the `Frame` class to determine what edges correspond to each of the input variables, and it updates the location of any variables that are assigned in the statement.

The compiler accounts for interruptions in control flow such as break statements, continue statements, and return statements using temporary boolean variables. It creates two variables for each loop, one for continuing and one for breaking, as well as one variable for return statements. The boolean variables indicate whether to skip different portions of the code. Each variable is initialized to true at the beginning of the method. Whenever the compiler encounters a continue, break, or return statement, it sets each of the subsequent variables to false. When compiling the code, it checks each such variable at the proper time. As an example, this would result in the transformation suggested in Figure 24.

Original Code:

```
int foo(int[][] array) {
    int sum = 0;
OuterLoop: for(int i = 0; i < array.length; i++) {
        for(int j = 0; j < array[i].length; j++) {
            if (array[i][j] < 0)
                break OuterLoop;
            sum++;
            if (array[i][j] == 0)
                return 0;
            sum++;
            if (array[i][j] == 10)
                continue;
            sum += array[i][j];
        }
    }
    return 1;
}
```

Transformed Code:

```
int foo(int[][] array) {
    boolean continueVarInner = true;
    boolean breakVarInner = true;
    boolean continueVarOuter = true;
    boolean breakVarOuter = true;
    boolean returnVar = true;
    int returnValue = -1;
    int sum = 0;
    for(int i = 0; breakVarOuter && i < array.length; i++) {
        for(int j = 0; breakVarInner && j < array[i].length; j++) {
            if (array[i][j] < 0) {
                continueVarInner = false;
                breakVarInner = false;
                continueVarOuter = false;
                breakVarOuter = false;
            }
            if (continueVarInner) {
                sum++;
                if (array[i][j] == 0) {
                    returnValue = 0;
                    continueVarInner = false;
                    breakVarInner = false;
                    continueVarOuter = false;
                    breakVarOuter = false;
                    returnVar = false;
                }
                if (continueVarInner) {
                    sum++;
                    if (array[i][j] == 10)
                        continueVarInner = false;
                    if (continueVarInner)
                        sum += array[i][j];
                }
            }
        }
    }
    if (returnVar)
        returnValue = sum;
    return returnValue;
}
```

Figure 24: Transformation Performed To Eliminate Interruption of Control Flow

While this code would be relatively inefficient if executed directly due to the extensive use of temporary variables, optimization ought to eliminate this problem. In particular, the

unused variable `continueVarOuter` would be eliminated during optimization, while `returnVar` and `breakVarOuter` would be combined into a single variable. After this optimization, the compiler could correct inefficiencies introduced by the remaining temporary boolean variables when the data flow graph is converted into Fresh Breeze machine code. In general, this conversion process could look for occasions when variables contain known constants and replace them with the appropriate branching instructions.

In some cases, it is not yet known whether a given variable is storing an object or an array. It is helpful to compute this information, in order to open up potential opportunities for optimization. To this end, before beginning to compile a data flow graph, the compiler computes the type of each object or array variable in the intermediate representation at each location in the code. To do so, it uses the `doFixedPointMap` algorithm described in section 3. Each statement that uses a variable as an array generates a mapping from that variable to the array datatype, while each statement that uses a variable as an object generates a mapping from that variable to the object datatype.

Compiling the condition of an if statement takes special care, as the condition may consist of more than one node in the control flow graph. Figure 25 contains an example of an if statement whose condition consists of multiple CFG nodes, along with a possible DFG representation of the condition.

Code:

```
a || (b && c)
```

Data Flow Graph:

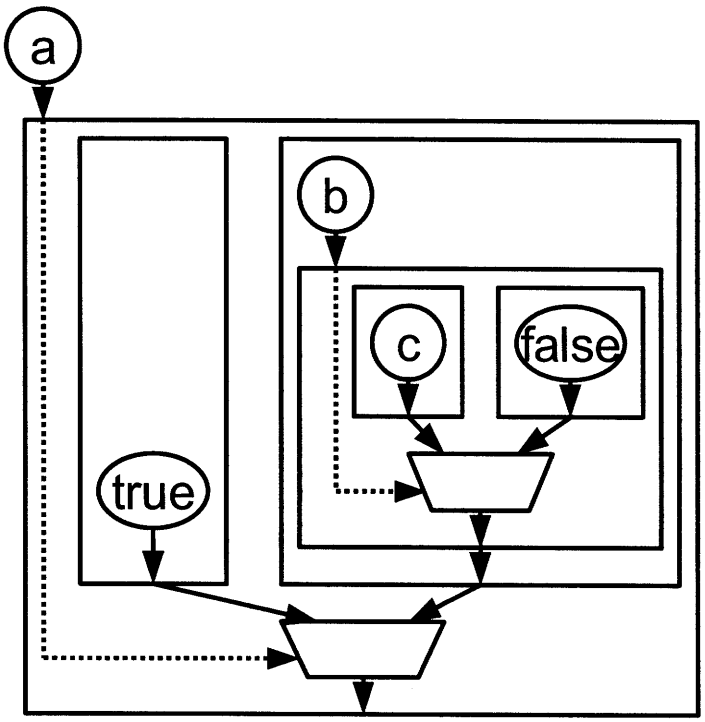

Figure 25: Data Flow Graph Representation of a Short-Circuiting Condition

Note that as it stands, the above data flow graph is less efficient than the original program, since it makes use of temporary variables to store boolean values. However, any such inefficiencies could be corrected when the data flow graph is converted into Fresh Breeze machine code by substituting branching in place of temporary booleans. This would allow the compiler to recover the original, branch-based approach to the computation.

To compile a given if statement's condition, the compiler creates temporary variables for

each of the nodes in the condition other than the first. Each variable represents whether the program should execute the node. It sets the values of these variables to be false in the current `Frame`. It also creates a temporary variable representing whether it will jump to the first branch or the second branch. It creates a new `Frame` and compiles the first node in the condition by a call to `compileNode`. Then, the compiler updates the appropriate temporary variables within the context of the new `Frame`. For example, say that E is the edge indicating whether to jump to the first successor or to the second successor of the node. Also, suppose that if this value is true, the program jump to a node X, which is not one of the branches, and if it is false, it jump to the "true" branch of the if statement. In this case, the compiler would update the condition for X to be the edge E, and it would update the condition for the if statement to be the negation of this edge.

Having compiled the first node in the condition, it proceeds to the second node. The second node is only executed if its temporary variable is true, so it needs to create a `DfgConditionalGraph` for the node. The compiler creates a new `Frame` for the second node and compiles it with another call to `compileNode`. Then, the compiler updates the appropriate temporary variables based on the result of that node's condition, as it did with the first node. Because the locations of these variables are updated in the context of the new frame, changes to such variables will naturally be manifested in the `DfgConditionalGraph`'s merge nodes, with one input to each of the merge nodes flowing from the old value of the variable and one flowing from the new value. The compiler continues compiling the remaining nodes in the condition in the same manner. Finally, it can compile the body of the if statement into a `DfgConditionalGraph`, using as the condition the temporary variable for the if statement.

5. Checking the Aliasing Restrictions

The compiler is designed to check the input programs to ensure that they meet the aliasing restrictions detailed in section 2.1. Checking the seventh aliasing rule, which forbids == and != under certain circumstances, is a simple matter of scanning the intermediate representation of the program for illegal uses of the operators. The remaining checks require the compiler to make use of information regarding the ancestors of objects and arrays. The checks are performed as it compiles a method, since it already has the necessary information regarding objects' and arrays' ancestors.

To check aliasing rule 1, which forbids any argument to a method call from being a descendant of any other argument, the compiler performs the necessary check whenever compiling a method call. For each argument, it checks the argument and each of its ancestors to see if its `ClassReference` matches the `ClassReference` of any of the other arguments. To check for violations of rule 2, whenever the compiler encounters a return statement, it checks to see whether the return value is a descendant of any of the arguments to the method. If a walk up the tree of ancestors reveals that this is the case, it raises an exception. To prevent violations of rule 3, which only allows a programmer to set an object's parent if it is known to have no parent, the compiler performs a check whenever an array's element or an object's field is being set. If the object to which it is being set has a known parent or is an argument to the method, it fails the check.

Rules 4, 5, and 6 are a bit trickier, as they require the compiler to ensure that certain objects and arrays are not accessed or altered. In order to check for violations of these rules, in

each `Frame`, it keeps track of which `ClassReferences` may not be accessed and which may not be altered. Each time it compiles a statement that accesses or alters an object or array, it checks the `Frame` to see whether it and all of its ancestors may be accessed or altered; if not, the compiler raises an exception.

In order to disallow altering an object whose parent depends on which branch was taken, in violation of rule 4, the compiler needs to be sure to mark such `ClassReferences` as unalterable. Whenever it finishes compiling a `DfgConditionalGraph`, it checks each `ClassReference` that was changed by one of the branches' `Frames` to see whether it has the same parent in each branch. For each one whose parent is ambiguous, it indicates that the `ClassReference` may not be altered. Also, whenever the compiler encounters a statement setting an array element or an object's field to some array or object, it indicates that the `ClassReference` may subsequently be altered, since its parent is now known.

To prevent violations of rule 5, whenever an array or object is altered, the compiler find the highest array ancestor of the array and mark each of its children as unable to be accessed or altered, aside from the one that is an ancestor of the array or object that is being altered. For rule 6, whenever an array is altered, it goes through its children and mark them as unalterable.

There are some aliasing violations that are not caught in a single pass of compilation. These cases arise when there is looping. Consider, for example, the method in Figure 26.

```
void foo(Foo foo, Foo foo2) {
    for(int i = 0; i < 10; i++) {
        if (i == 5)
            foo = foo2;
        else
            foo.bar = new Bar();
    }
}
```

Figure 26: An Aliasing Violation in a Loop

This code violates rule 4 because in the statement `foo.bar = new Bar()`, the identity of the parent of `foo` is unknown. However, a single pass through the loop will not be able to identify this violation. To solve this problem, the compiler compiles each loop twice and discarding the results of the second compilation. The second pass through the above loop would be aware that the parent of `foo` is unknown, so it would reject the method. This solution, while inelegant and inefficient, is exceedingly simple, requiring only three lines of code. Note that there is undoubtedly a more elegant but substantially more complex solution.

6. Optimization

6.1. Overview of Optimization

The compiler performs several optimizations on the data flow graph. These optimizations include classical optimizations, which are commonly performed on control flow graphs, optimizations that are specific to data flow graphs, optimizations that are specific to Java, and optimizations designed for the Fresh Breeze memory model.

Each optimization is implemented as a transformation that modifies a data flow graph in

place. The compiler performs the optimizations on the data flow graph for each method in no particular order. It repeatedly applies them to a given DFG so that opportunities for optimization that are uncovered by one optimization may be utilized by another optimization. It repeats the optimizations until a fixed point is reached; that is, until each optimization indicates that it cannot optimize the DFG any further.

Some optimizations do not exactly preserve the behavior of the original program, due to imprecision and special cases regarding floating point numbers. For example, code is optimized in the form shown in Figure 27:

Original Code:

```
float foo(float x, float y, float z) {
    return (x + y) * (x + z + y);
}
```

Optimized Code:

```
float foo(float x, float y, float z) {
    float temp = x + y;
    return temp * (temp + z);
}
```

Figure 27: An Imprecise Floating Point Optimization

The optimization slightly alters the results of the method, since due to imprecision in floating point numbers, `x + z + y` is not exactly equivalent to `x + y + z`. Because such optimizations alter methods' results, a compiler flag enables the user to disable them (more

precisely, to alter the optimizations' behavior).

Some of the optimizations make use of `DfgCommutativeOpNodes`. Such a node represents an operation that is commutative and associative, such as addition and xor. Unless the user sets the compiler flag mentioned previously, floating point addition and multiplication are treated as commutative and associative, although such operations are not associative, strictly speaking. `DfgCommutativeOpNodes` are helpful because, for example, they make it more obvious that `x + y + (-x)` can be simplified to `y`, where `x` and `y` are both integers.

Before performing any optimizations, the compiler does a transformation that is designed to reveal opportunities for optimization. It replaces each subtraction operation `x - y` with the addition `x + (-y)` and, if imprecision in floating point operations is permitted, each floating point division `x / y` with the multiplication `x * (1 / y)`, where `1 / y` is treated as a unary operation on y. Both of these transformations allow the compiler to combine several operations into a single `DfgCommutativeOpNode`, making it easier to identify occasions for optimization. For example, these transformations will make it easy to transform `x / y / z` into `x / (y * z)`, saving an expensive division operation.

6.2. Classical Optimizations

One of the optimizations the compiler uses is dead code elimination. To perform dead code elimination, it identifies every needed node in the graph by performing a search from each sink node. For each node N, it checks the nodes that have an output of N as an input. It discards each such node, causing them and each node that uses their results to be garbage collected at

some point in the future. This optimization takes linear time.

Constant folding is another classical optimization the compiler performs on data flow graphs. To do constant folding, it finds each node in the graph that performs a binary or unary operation and that has constant inputs. It simplifies each such node by substituting the appropriate constant in place of its output. For example, such an optimization would replace the output to 3 + 7 with the constant 10. This also takes linear time.

The compiler also performs algebraic simplification on the data flow graph. Algebraic simplification is essentially a set of pattern matching rules that allow the compiler to simplify a data flow graph. One simplification is the removal of the identity from addition, multiplication, bitwise and, bitwise or, and xor operations. For example, the expression 0 + x is simplified to x. Another simplification is the removal of redundant promotions or demotions. For example, itob(btoi(true)) is simplified to true. This simplification eliminates some inefficiency introduced when converting the Java bytecode into its intermediate representation. The compiler performs several additional simplifications that are not described here because they are so numerous and simple. The optimization takes $O(n)$ time.

Loop-invariant code motion provides further opportunities for optimizing a data flow graph. The optimization entails identifying nodes in a loop's body that flow directly from the loop-invariant inputs to the graph, rather than from loop-dependent variables, and hoisting such nodes out of the loop. The optimization is relatively easy to implement, using only about 200 lines of code, and operates in linear time.

Another possible optimization is method inlining. To do inlining, one would first identify occasions in which one should inline a method using some sort of heuristic. Then, one would

copy the graph for each method that one is inlining and insert it in place of each method call node where it is being inlined. Inlining takes linear time.

Common subexpression elimination enables the compiler to get rid of repeated computations of the same value. For the most part, this is straightforward. The compiler builds a map from expressions to nodes and use it to identify multiple occurrences of the same expression. When it finds a repeated expression, it retains only one occurrence of the expression and have its output flow to the other places where the expression's result is used. This approach allows the compiler to eliminate operations as diverse as additions, method calls, checking the type of a class, and entire subgraphs of a method.

The use of `DfgCommutativeOpNodes` allows the compiler to make use of more advanced common subexpression elimination. If two `DfgCommutativeOpNodes` have the same operation and have at least two identical inputs, it can simplify them by computing some portion of the commutative operation a single time. For instance, the compiler can perform the optimization suggested in Figure 28.

Original Code:

```
int x = a + c + b;
int y = a + b + d;
```

Optimized Code:

```
int temp = a + b;
int x = temp + c;
int y = temp + d;
```

Figure 28: Optimizing Common Subexpressions

For each commutative and associative operation, the algorithm maintains a priority queue of the `DfgCommutativeOpNodes` for that operation sorted by the number of remaining inputs. It retrieves the node X with the most such inputs and compute the `DfgCommutativeOpNode` Y that has the most inputs in common with it. If there is an overlap of more than one input, it creates a new `DfgCommutativeOpNode` Z whose inputs are the overlapping inputs. It removes the overlapping inputs from X and Y and replace them with the output of Z.

While this approach works well in typical cases, it is not optimal, relying on heuristics in an attempt to get relatively good results. It is worth considering how one might further reduce redundant computations.

In the worst case, for each repeated computation of a particular value, common subexpression elimination will create a number of edges that is linear in the size of the program. For this reason, the optimization has a quadratic worst-case running time.

6.3. Other Optimizations

Several of the optimizations used by the compiler are specific to data flow graphs. One such optimization is "constant merging", an optimization that operates on `DfgConstantNodes`,

which output fixed constants. It takes `DfgConstantNodes` that output the same constant and replaces them with references to one `DfgConstantNode`. While this optimization has no effect on the speed of the resulting program, it reduces the space used by the data flow graph, makes it faster to perform future optimizations, and makes it easier to read output representing a data flow graph. The optimization takes $O(n)$ time.

In the "graph removal" optimization, the compiler locate "free-standing" subgraphs; that is, graphs other than conditional graphs and loop graphs and those contained in conditional graphs or loop graphs. It removes each such subgraph by moving its nodes to the graph in which the subgraph lies. Again, this does not affect the speed of the compiled method, but does make it easier to work with the data flow graph, at the expense of linear running time.

In the "graph port elimination" optimization, the compiler removes graphs' input and output ports and convert loop-dependent variable ports into loop-independent variable ports. To perform the optimization, it goes through each of the method's subgraphs. It removes each of the output ports that does not flow into some other node. The compiler checks each input port to each graph's sink node to find pairs of input ports that flow from the same output, and it removes one port from each such pair. In addition, it removes input ports to each graph's sink node that flow directly from the source node.

To simplify loop graphs, the compiler looks for pairs of loop-dependent variables that are equivalent; that is, pairs of loop-dependent variables that are initialized from the same output and that are set to the same output value at each iteration. It retains only one variable port in each such pair. Also, it finds loop-dependent variable ports that are, in fact, loop-independent. Such ports are those that flow directly from an output port of the source node of the loop body's graph

to the corresponding input port in the sink node. It makes these nodes instead flow directly from the loop graph's source node to the loop body's graph. Lastly, the compiler remove loop-dependent variables that are never used, which are those variables that do not flow to any nodes from the loop body's source node or from the loop graph itself. Loop port elimination has linear running time.

There are also several optimizations one can perform on conditional graphs in linear time. If the condition of such a graph is a constant, the compiler can replace the conditional graph with the appropriate subgraph. It can also simplify conditional graphs whose condition flows from a node that inverts its boolean input by reversing the true and false subgraphs and changing the condition to the input to the inverting node. If the graph's condition flows into one of the nodes in one of the conditional graphs subgraphs, the condition's value is known to be true in the case of the left subgraph and false in the case of the right subgraph, so the compiler can substitute the appropriate boolean constant. It can replace any output to the conditional graph that is the same in both branches with its value in the two branches. Any other output to the graph that is a boolean can be replaced with either a reference to the graph's condition or to the inverse of the graph's condition. The compiler can also simplify pairs of conditional graphs whose conditions are the same or are opposites of each other to a single conditional graph.

Furthermore, the compiler can eliminate operations that are performed on conditional graphs' outputs whose values are constant, regardless of which branch is taken. For example, it can perform the optimization suggested in Figure 29.

Original Code:

```
if (a) {
    b = 1;
    c = 2;
}
else {
    b = 3;
    c = 4;
}
d = b + c;
```

Optimized Code:

```
if (a) {
    b = 1;
    c = 2;
    d = 3;
}
else {
    b = 3;
    c = 4;
    d = 7;
}
```

Figure 29: Optimizing a Conditional Statement

The compiler can perform a few optimizations that rely on specifics of the Java language. Whenever a program retrieves the length of an array that was created within the method, one can simply use the length that was supplied when creating the array. More generally, if a variable may store one of multiple arrays created in the method and all of these arrays were created with the same length, the program may use that length directly rather than retrieving the array's length. Wherever a program retrieves an element of an array that was just created, this value is known to be 0 (or false or null), provided that the index of the array is known not to be out of

bounds, so the program can use the appropriate constant value instead.

Also, the compiler can use known information about an object's type. If a program checks whether a variable is an instance of a particular object type, the compiler can check all possible types of objects that may be stored in that variable. If they all match the desired type, it can replace the check with a check for null. If they all fail to match the desired type, it can replace the check with the constant "false". Since `java.lang.Object` is the superclass of every object, it also replaces checking whether a given object is an instance of `java.lang.Object` with checking whether the object is null. Each of these optimizations runs in linear time.

Other optimizations are helpful for the write-once memory model of the Fresh Breeze architecture. Particularly useful in this regard is the "repeated set elimination" optimization. The optimization replaces repeated setting a given array's element a given object's field to a single setting. For example, this would convert the statements `array[x] = 0; array[x] = 1;` into the single statement `array[x] = 1`.

This optimization is particularly helpful when objects with ancestors are altered. Consider the code in Figure 30.

```
int[] x = y[0][1][2];
x[0] = 1;
x[1] = 17;
x[2] = 4;
```

Figure 30: Repeated Set Elimination

In a naïve approach, an altered copy of `y` is created every time `x` is changed. Repeated set

elimination transforms a graph that uses the naïve approach to one that creates a single altered copy of `y`.

To perform repeated set elimination, the compiler needs to identify groups of edges that represent the "same" object or array. This process is a matter of finding pairs of edges with the "same" object or array and using an algorithm to use these pairs to compute a set of edges for each given object. For example, the output of a `DfgArraySetNode` is the "same" array as the input array, so these edges would comprise one such pair.

Having identified the relevant objects and arrays, the compiler needs to determine which array elements and object fields are "live" at which points in the code. Live elements and fields are understood to be elements and fields that may be used before they are reassigned. Note that elements and fields may be live at some points of the graph and dead at other points. Determining which elements and fields are live is essential to ensure correct behavior, as in the method in Figure 31.

```
void foo(int[] array) {
    array[0] = 5;
    int x = array[0];
    array[0] = 10;
    return x;
}
```

Figure 31: Usage of Live Elements

In the method `foo`, it is not permissible to eliminate the statement `array[0] = 5`. This is because the `array[0]` is live, since it is accessed in the statement int `x = array[0]` before it is reassigned in the statement `array[0] = 10`. More generally, eliminating setting an

object's field or an array's element is only permissible if the field or element is dead.

To compute the liveness of each element and field at any given point in the graph, the compiler relies on fixed-point computation, using the algorithm described in the appendix. As in live variable analysis for control flow graphs, the computation goes through the graph backwards. For each node, it identifies the live fields and elements that are "generated," which are the fields and elements that are used by the node, and the live fields and elements that are "killed," which are the fields and elements that are reassigned by the node. The compiler represents a given object's field using the group of edges that correspond to that object and the particular field the program is retrieving. Likewise, it represents an array element using the group of edges corresponding to the array and the edge representing index of the element in question. For the fixed-point computation, the compiler also needs to indicate that every field and element of each return values is live at the sink node of the method's data flow graph, as are all of their descendants. Then, it uses the ordinary algorithm for fixed point computation presented in the appendix to determine liveness. While the algorithm is dealing with more nodes than usual, since each operation rather than each basic block represents a single node, the algorithm is still correct.

Due to a bit of sloppiness in the above approach, the algorithm misses some opportunities for optimization. For instance, consider the method in Figure 32.

```
void foo(int[][] x) {
    int[] y = x[0];
    y[0] = 5;
    x[0] = y;
    y[0] = 10;
}
```

Figure 32: An Opportunity for Repeated Set Elimination

In this case, it is possible to remove the assignment `y[0] = 5`. However, the algorithm takes a conservative approach, labeling the elements of `y` as live immediately before the statement `x[0] = y`. With some modification that entails keeping track not only of arrays' immediate elements, but also of their elements' elements and so on, the algorithm could be designed to identify this and similar occasions for optimization.

Repeated set elimination takes $O(n^5)$ running time, where n is the size of the program being compiled, because of its usage of the fixed-point algorithm. The universe of values provided to the algorithm has a size $O(n)$. The fraction of iterations in the fixed-point algorithm that have an effect is 1 in $O(n)$, since the size of the `changed` variable is at most $O(n)$ and it decreases by one each time that an iteration has no effect. At each iteration of the fixed-point algorithm that has an effect, the number of live elements and fields associated with a given node increases by at least one. Since there are at most $O(n)$ nodes, each of which may have a total of $O(n)$ live elements and fields, there will be a quadratic number of iterations that have an effect. Each iteration must process each of the $O(n)$ nodes. Doing so requires taking the union of two bit sets, which takes linear time. Multiplying these values gives a running time of $O(n^5)$. However, the fixed-point algorithm is generally accepted to run quickly in practice. The remainder of the repeated set elimination optimization takes linear time.

Another transformation opens further opportunities for optimization. The transformation has to do with multiple references to the "same" array or object. Consider, for example, the code in Figure 33.

```
int a = x[0];
x[1] = 5;
int b = x[0];
```

Figure 33: Repeated Access of a Single Array Object

One can optimize the last statement to `int b = a`. However, this optimization is not immediately apparent when looking at the data flow graph. This is because in the last statement, `x` refers to the output of the `DfgArraySetNode` for the statement `x[1] = 5`. To make this optimization more apparent, the compiler attempts to move array accesses and the like "higher" in the graph. In this case, the compiler could move the array access in the last statement so that the array input is the same as the array input in the first statement. Now that there are two `DfgArrayGetNodes` with the same two input edges, common subexpression elimination can identify the array access as a common subexpression, allowing the compiler to optimize the last statement. More generally, the goal is to move accessing an array's element or an object's field, retrieving the length of an array, comparing an array or object with null, and checking the type of an object as "high" in the graph as possible, so that the compiler uses the "highest" possible array or object edge as input.

To find these edges, the compiler relies on a relatively simple subprocedure `sameEdge` that identifies "local" edges that hold the "same" array or object as a given edge and are higher than the given edge. For example, if the compiler is looking at the output to a

`DfgArraySetNode`, it know that the output edge holds the "same" array as the input array. If it is looking at the output to the source node of a loop body, it knows that the edge holds the "same" array or object as the corresponding input to the body's sink node (the loop variable's value at the end of an iteration) and the input to the body (the loop variable's initial value).

The aim is to find the earliest possible "source" of a given object or array. Possible sources include outputs to nodes that create new objects or arrays and the return values of methods.

Accessing an array's element or an object's field are treated differently from the other operations because elements and fields are not constant throughout an object's lifetime. In this case, the compiler treat the outputs of `DfgArraySetNodes` and `DfgSetFieldNodes` as potential sources. For these two particular operations, it aims to find the earliest single source for the array or object in question. Note that it is possible for an edge for an object or array to have multiple potential sources; in this case, there is no higher edge that the compiler may use in its place. For example, consider the code in Figure 34.

```
void foo(int[] x) {
    if (x.length > 2)
        x[2] = 2;
    else
        x = new int[3];
    x[0] = 0;
    x[1] = 1;
    return x[2];
}
```

Figure 34: Identifying Source Edges

In the statement **return** `x[2]`, the compiler initially use the output of the statement

`x[1] = 1` as the array input. It would be preferable to use a higher edge in place of this input. The compiler is permitted to use the output to `x[0] = 0` in its place. However, it may not use the output to `x[2] = 2` or `x = new int[3]`, because neither is the sole source of the array in question. As such, the compiler needs an algorithm that identifies the output to `x[0] = 0` as the correct edge to use.

The algorithm uses dynamic programming to identify the sole source of a given object or array. If `sameEdge` returns a source edge, it returns that edge. Otherwise, it recurses on each of the edges the subprocedure returned. If each gives the same source edge, it returns that edge. Otherwise, if the current edge is the output to a `DfgArraySetNode` or a `DfgSetFieldNode`, it returns the edge. Otherwise, it returns null, indicating that there is no single source edge.

When dealing with operations other than getting an array's element or an object's field, it is simpler to identify what source to use. In this case, the compiler does not need to find the sole source of a given object or array; it may use any source edge. Now rather than following several edges to their sources, the compiler select a single "best" edge to follow for each given node. For example, if it is considering the output to a loop body's source node, it follows the input to the body (the loop variable's initial value) while ignoring the corresponding input to the body's sink node (which contains the loop variable's value at the end of an iteration).

This optimization has quadratic running time in the worst case. This is because identifying the source of a given edge, an operation that is performed a linear number of times, takes a linear amount of time.

7. Future Work

While this project entailed much work on the Fresh Breeze compiler, future work could build on this. For one, under the current implementation, the optimization process is particularly slow, taking about a minute to optimize the 17,000 lines of code in test cases. It is worth considering how its speed could be improved. Although most or all of the algorithms the compiler uses are linear or quadratic, it is likely that at least some of their asymptotic running times could be improved. Also, the intermediate representation of the program is particularly sloppy, using lots of temporary variables. Perhaps performing some basic optimization on the intermediate representation, such as copy propagation and dead code elimination, before compiling to a data flow graph would save time on the whole. In addition, rather than applying the optimizations in an arbitrary order, one could perhaps order them more efficiently. Profiling the program's execution may reveal additional ways to improve speed without substantial algorithmic changes.

Some of the optimization algorithms can be improved to produce better data flow graphs. For example, the algorithms the compiler uses for repeated set elimination and common subexpression elimination are known to be suboptimal. Details regarding how these algorithms might be improved are suggested in sections 6.1 and 6.3.

While the compiler is quite successful at producing an optimized data flow graph representation of a given method, it does not yet target Fresh Breeze machine code. Future work could focus on developing a code generator to convert the DFGs to machine code.

Additional work could design the compiler to reveal opportunities for further optimizations. In particular, work could be done to deal more efficiently with arrays, which due

to the constraints of the Fresh Breeze memory model each have to be stored as a tree rather than as a single linear block of memory. To this end, one could add special nodes to identify occasions on which an array's elements are set in order. This would allow these operations to be compiled in such a manner that one does not have to walk the entire depth of the tree each time an element is set. Furthermore, one could introduce nodes that indicate when successive iterations of a loop are independent of each other. By identifying independent iterations, loops could be divided into several pieces, each of which is worked on by a separate processor or core.

The current implementation of the compiler does not handle exceptions, including trying to access an element of an array that is out of bounds. Also, the compiler will get rid of loops whose results are not used. This is technically incorrect, since the loop might be an infinite loop. Future work could add support for exceptions and properly handle such infinite loops. One possible implementation of these exceptional cases would be to have special exception edges as outputs to nodes that might throw exceptions and loops that might be infinite loops.

Appendix A: Pseudocode Listings

A.1. Fixed Point Computation

```
Map<N, Set<V>> doFixedPointUnion(Map<N, Set<V>> initialValues, Map<N, Set<V>>
generated, Map<N, Set<V>> killed, Map<N, Set<N>> successors)
    Map<N, Set<N>> predecessors
    For (node, nodes) ∈ successors
        For node2 ∈ nodes
            predecessors[node2] = predecessors[node2] ⋃ {node}

    Map<N, Set<N>> in
    Map<N, Set<N>> out
    Set<N> changed
    For (node, succ) ∈ successors
        If initialValues[node] = null
```

```
            changed = changed ∪ {node}
            out[node] = generated[node]
        Else
            in[node] = initialValues[node]
            out[node] = in[node] ∪ generated[node] - killed[node]

    While changed ≠ {}
        N node = Any element in changed
        changed = changed - {node}
        Set<V> newIn
        For node ∈ predecessors[node]
            newIn = newIn ∪ out[node]

        If newIn ≠ in[node]
            in[node] = newIn
            out[node] = newIn ∪ generated[node] - killed[node]
            For node2 ∈ successors[node]
                changed = changed ∪ {node2}
    Return in
```

A.2. Identifying Loops

```
Boolean canReach(Node start, Node goal, Set<Node> nodes, Set<Node> &visited,
Set<Node> &canReach, Set<Node> &cantReach)
    If start ∈ canReach
        Return true
    Else If start ∈ cantReach
        Return false

    Boolean r
    If start ∉ nodes or start ∈ visited
        r = false
    Else
        visited = visited ∪ {start}
        If start = goal
            r = true
        Else
            r = false
            For succ ∈ start.successors
                If canReach(succ, goal, nodes, visited, canReach, cantReach)
                    r = true
                    Break

    If r
        canReach = canReach ∪ {start}
    Else
        cantReach = cantReach ∪ {start}
    Return r


Boolean checkDeadEnd(Node node, Set<Node> nodes, Set<Node> &visited)
```

```
    If node ∉ nodes
        Return false
    If node ∈ visited
        Return true
    visited = visited ⋃ {node}
    For succ ∈ node.successors
        If Not checkDeadEnd(succ, nodes, visited)
            Return false
    Return true


Boolean checkDeadEnd(Node node, Set<Node> loopNodes, Set<Node> nodes,
Set<Node> &visited)
    For pred ∈ node.predecessors
        If pred ∉ loopNodes
            Return false
    If checkDeadEnd(node, nodes, visited)
        For node2 ∈ visited - {node}
            For pred ∈ node2.predecessors - visited
                Return false
        Return true
    Else
        Return false


List<Node> computeLoopNodes(Node start, Set<Node> nodes, Set<Node> &deadEnds)
    Set<Node> loopNodes
    Set<Node> canReach
    Set<Node> cantReach
    For node ∈ nodes - {start}
        If canReach(node, start, nodes, {}, canReach, cantReach)
            loopNodes = loopNodes ⋃ {node}

    Set<Node> loopNodes2 = loopNodes
    For node ∈ loopNodes
        For succ ∈ node.successors - loopNodes
            Set<Node> visited
            If checkDeadEnd(succ, loopNodes, nodes, visited)
                loopNodes2 = loopNodes2 ⋃ visited
                deadEnds = deadEnds ⋃ {succ}

    Set<Node> loopNodes3 = {start}
    For node ∈ loopNodes2 - {start}
        If node.predecessors - loopNodes2 = {}
            Add node to loopNodes3
    Return loopNodes3


List<Group> topologicalSort(Set<Group> groups, Map<Group, Set<Group>>
successors)
    Map<Group, Int> numPredecessors
    For group ∈ groups
        For succ ∈ successors[group]
            numPredecessors[succ] = numPredecessors[succ] + 1
```

```
    Set<Group> remainingGroups = groups
    PriorityQueue<Group> sortedGroups sorted by predicate(group1, group2)
        If numPredecessors[group1] < numPredecessors[group2]
            Return group1
        Else If numPredecessors[group1] > numPredecessors[group2]
            Return group2
        Else
            Return arbitrary comparison of group1 and group2
    sortedGroups = groups

    List<Group> groups2
    While sortedGroups ≠ {}
        Group group = First element in sortedGroups
        Add group to groups2
        sortedGroups = sortedGroups - {group}
        For group2 ∈ successors[group]
            numPredecessors[group2] = numPredecessors[group2] - 1
    Return groups2


List<Node> orderBeforeFix(Set<Node> nodes, Node start, Map<Node, Node>
&loopStartToLoopEnd, Set<Node> &deadEnds, Boolean isInLoop)
    Set<Node> remainingNodes = nodes
    Set<Node> seen
    If isInLoop
        seen = start.successors ∩ remainingNodes
        remainingNodes = nodes - {start}
    Else
        seen = {start}

    Map<Node, Int> numPredecessors
    For node ∈ remainingNodes
        numPredecessors[node] = |node.predecessors|

    PriorityQueue<Node> sortedNodes sorted by predicate(node1, node2)
        If node1 ∈ seen and node2 ∉ seen
            Return node1
        Else If node1 ∉ seen and node2 ∈ seen
            Return node2
        Else If numPredecessors[node1] < numPredecessors[node2]
            Return node1
        Else If numPredecessors[node1] > numPredecessors[node2]
            Return node2
        Else
            Return arbitrary comparison of node1 and node2
    sortedNodes = remainingNodes

    Set<Group> groups
    Map<Node, Group> nodeToGroup
    While sortedNodes ≠ {}
        Node node = First element in sortedNodes
        Group group
        If numPredecessors[node] = 0 or node.successors = {}
            group.nodes = {node}
```

```
        Else
            group.nodes = computeLoopNodes(node, remainingNodes, deadEnds)
        groups = groups ∪ {group}
        For node ∈ group
            sortedNodes = sortedNodes - {node}
            remainingNodes = remainingNodes - {node}
            nodeToGroup[node] = group
        For node ∈ group
            For succ ∈ node.successors ∩ remainingNodes
                numPredecessors[succ] = numPredecessors[succ] - 1
                seen = seen ∪ {succ}

    Map<Group, Set<Group>> groupSuccessors
    For (node, group) ∈ nodeToGroup
        For succ ∈ node.successors
            If nodeToGroup[succ] ≠ group
                groupSuccessors[group] = groupSuccessors[group] ∪
{nodeToGroup[succ]}

    List<Group> groups2 = topologicalSort(groups, groupSuccessors)
    List<Node> nodes2
    If isInLoop
        Add start to nodes2
    For group ∈ groups2
        If |group.nodes| = 1
            Add group.nodes to nodes2
        Else
            List<Node> loopNodes = orderBeforeFix(group.nodes,
group.nodes[0], loopStartToLoopEnd, deadEnds, true)
            loopStartToLoopEnd[loopNodes[0]] = loopNodes[|loopNodes| - 1]
            Add loopNodes to nodes2
    Return nodes2


void computeLoopInfo(List<Node> nodes, Map<Node, Node> loopStartToLoopEnd,
Map<Node, Node> &enclosingLoops, Map<Node, Node> &loopParents)
    Stack<Node> loopStartStack
    Stack<Node> loopEndStack
    For node ∈ nodes
        If loopStartToLoopEnd[node] ≠ null
            loopParents[node] = Top of loopStartStack
            Push node onto loopStartStack
            Push loopStartToLoopEnd[node] onto loopEndStack
        enclosingLoops[node] = Top of loopStartStack
        While node = Top of loopEndStack
            Pop from loopStartStack
            Pop from loopEndStack


Node lowestCommonAncestor(Set<Node> nodes, Map<Node, Node> parents)
    Node lowest = Any element in nodes
    nodes = nodes - {lowest}
    Set<Node> visited = {lowest}
    For node1 ∈ nodes
```

```
        Node node2 = lowest
        Set<Node> visited2
        While node1 ≠ null or node2 ≠ null
            If node1 ≠ null
                If node1 ∈ visited
                    Break
                visited = visited ⋃ {node1}
                node1 = parents[node1]
            If node2 ≠ null
                node2 = parents[node2]
                If node2 ≠ null
                    If node2 ∈ visited
                        lowest = node2
                        visited = visited ⋃ visited2
                        Break
                    visited2 = visited2 ⋃ {node2}
        If node1 = null and node2 = null
            Return null
    Return lowest


Set<Node> deadEndLoopAncestors(Set<Node> nodes, Map<Node, Node> parents)
    Node node = lowestCommonAncestor(nodes, parents)
    Set<Node> ancestors
    For node2 ∈ nodes
        Node node3 = node2
        While node3 ≠ null
            node3 = parents[node3]
            If node3 = node
                ancestors = ancestors ⋃ {node2}
                Break
    Return ancestors


List<Node> topologicalSort(Map<Node, Node> parents)
    Map<Node, Int> childrenLeft
    For (child, parent) ∈ parents
        childrenLeft[parent] = childrenLeft[parent] + 1
    PriorityQueue<Node> nodesLeft sorted by predicate(node1, node2)
        If childrenLeft[node1] < childrenLeft[node2]
            Return node1
        Else If childrenLeft[node1] > childrenLeft[node2]
            Return node2
        Else
            Return arbitrary comparison of node1 and node2
    For (child, parent) ∈ parents
        nodesLeft = nodesLeft ⋃ {child}

    List<Node> nodes
    While nodesLeft ≠ {}
        Node node = First element in nodesLeft
        nodesLeft = nodesLeft - {node}
        Add node to nodes
        Node parent = parents[node]
```

```
        If parent ≠ null
            childrenLeft[parent] = childrenLeft[parent] - 1
    Return nodes


Node firstNode(Node start, Set<Node> goals, Boolean isFirstCall)
    If Not isFirstCall and start ∈ goals
        Return start
    For succ ∈ start.successors
        Node node = firstNode(succ, goals, false)
        If node ≠ null
            Return node
    Return null

void findReachableNodes(Node start, Node goal, Set<Node> &visited)
    If node ∉ visited ∪ {goal}
        visited = visited ∪ {node}
        For succ ∈ start.successors
            findReachableNodes(succ, goal, visited)


List<Node> expand(Node node, Node goal, Map<Node, Integer> nodeIndices)
    Set<Node> visited
    findReachableNodes(node, goal, visited)
    Return visited sorted by predicate(node1, node2)
        If nodeIndices[node1] < nodeIndices[node2]
            Return node1
        Else If nodeIndices[node1] > nodeIndices[node2]
            Return node2
        Else
            Return null


Boolean mergeAt(Node node, Node end, Node start, Int startIndex, Int
endIndex, Map<Node, Int> nodeIndices, Set<Node> &mergeNodes, Set<Node>
&visited, Boolean isFirstCall)
    If Not isFirstCall and node = start
        Return false
    If node ∈ mergeNodes ∪ visited
        Return true
    visited = visited ∪ {node}
    If nodeIndices[node] < startIndex or nodeIndices[node] > nodeIndex
        Return true
    For succ ∈ node.successors
        If Not mergeAt(succ, end, start, startIndex, endIndex, nodeIndices,
mergeNodes, visited, false)
            Return false
    mergeNodes = mergeNodes ∪ {node}
    Return true


Boolean mergeAt(Node start, Node node, Int endIndex, Map<Node, Int>
nodeIndices)
    Return mergeAt(start, node, start, nodeIndices[start], endIndex,
```

```
nodeIndices, {}, {}, true)


Map<Node, Node> computeLoopSuccessors(List<Node> nodes, Map<Node, Map<Node,
Node> loopStartToLoopEnd, Set<Node> deadEnds, Map<Node, Node> &loopParents,
Map<Node, Set<Node>> &breakNodesMissed)
    Map<Node, Node> enclosingLoops
    computeLoopInfo(nodes, loopStartToLoopEnd, enclosingLoops, loopParents)
    Map<Node, Int> nodeIndices
    For i ∈ [0, |nodes|)
        nodeIndices[nodes[i]] = i

    Map<Node, Set<Node>> breakNodes
    Map<Node, Set<Node>> deadEndNodeLoops
    For node ∈ nodes
        Node loop1 = enclosingLoops[node]
        For succ ∈ node.successors
            If succ ∉ deadEnds
                Node loop2 = enclosingLoops[succ]
                If loop1 ≠ loop2 and (loop2 = null or loop1 ≠
loopParents[loop2])
                    breakNodes[loop1] = breakNodes[loop1] ⋃ {succ}
            Else
                deadEndNodeLoops[succ] = deadEndNodeLoops[succ] ⋃ {loop1}

    Map<Node, Set<Node>> loopDeadEndNodes
    For (node, loops) ∈ deadEndNodeLoops
        If |loops| > 1
            For loop ∈ deadEndLoopAncestors(loops, loopParents)
                loopDeadEndNodes[loop] = loopDeadEndNodes[loop] ⋃ {node}

    Map<Node, Queue<Node>> unmatchedDeadEndNodes
    Map<Node, Node> loopSuccessors
    For loopStart ∈ topologicalSort(loopParents)
        Int endIndex = nodeIndices[loopStartToLoopEnd[loopStart]]
        Set<Node> bns = breakNodes[loopStart]
        If |bns| > 1
            SortedSet<Node> bns2 sorted by predicate(node1, node2)
                If nodeIndices[node1] < nodeIndices[node2]
                    Return node1
                Else If nodeIndices[node1] > nodeIndices[node2]
                    Return node2
                Else
                    Return null
            bns2 = bns
            Node first = First element in bns2
            Node next = firstNode(first, bns, true)
            Set<Node> bns3 = Elements in bns2 before next
            If |bns3| = 1
                Set<Node> predLoops
                For pred ∈ first.predecessors
                    predLoops = predLoops ⋃ {enclosingLoops[pred]}
                If |predLoops| > 1
                    bns3 = {}
```

```
            For node ∈ bns3
                Node pred = Any element in node.predecessors
                Node loop = enclosingLoops[pred]
                breakNodesMissed[loop] = breakNodesMissed[loop] ∪
expand(node, next, nodeIndices)
            bns = bns - bns3

        Node successor
        Queue<Node> deadEndNodes = unmatchedDeadEndNodes[loopStart] ∪
loopDeadEndNodes[loopStart]
        If |bns| = 0
            If |deadEndNodes| > 0
                successor = Dequeue from deadEndNodes
            Else
                successor = null
        Else If |bns| = 1
            Node node = Any element in bns
            If |deadEndNodes| > 0 and mergeAt(loopParents[loopStart], node,
endIndex, nodeIndices)
                successor = Dequeue from deadEndNodes
            Else
                successor = node
        Else
            Node node1 = Any element in bns
            Node node2 = Any element in bns - {node1}
            If nodeIndices[node1] < nodeIndices[node2]
                successor = node1
            Else
                successor = node2

        loopSuccessors[loopStart] = successor
        Node parent = loopParents[loopStart]
        If parent ≠ null
            unmatchedDeadEndNodes[parent] = unmatchedDeadEndNodes[parent] ∪
deadEndNodes
    Return loopSuccessors


Map<Node, Set<Node>> expandDeadEnds(Set<Node> deadEnds)
    Map<Node, Set<Node>> ds
    For d ∈ deadEnds
        Set<Node> visited
        findReachableNodes(d, null, visited)
        ds[d] = visited
    Return ds


Map<Node, Node> newLoopStartToEnd(List<Node> nodes, List<Node> oldNodeOrder,
Map<Node, Node> oldLoopStartToEnd, Map<Node, Node> loopSuccessors, Map<Node,
Node> loopParents, Map<Node, Set<Node>> deadEnds2, Map<Node, Set<Node>>
breakNodesMissed)
    Map<Node, Node> extraLoops
    For (loop, succ) ∈ loopSuccessors
```

```
        For d ∈ deadEnds[succ]
            extraLoops[d] = loopParents[loop]
    For (loop, nodes) ∈ breakNodesMissed
        For node ∈ nodes
            extraLoops[node] = loop
    Map<Node, Int> nodeIndices
    For i ∈ [0, |nodes|)
        nodeIndices[nodes[i]] = i
    Map<Node, Int> oldNodeIndices
    For i ∈ [0, |oldNodeOrder|)
        oldNodeIndices[oldNodeOrder[i]] = i

    Map<Node, Node> newLoopStartToEnd
    For (start, end) ∈ oldLoopStartToEnd
        Set<Node> loopNodes
        For i ∈ [oldNodeIndices[start], oldNodeIndices[end]]
            If extraLoops[oldNodeIndices[i]] = null
                loopNodes = loopNodes ⋃ {oldNodeIndices[i]}
        Int index
        For index ∈ [nodeIndices[start], |nodes|)
            If nodes[index] ∉ loopNodes
                Node loop = extraLoops[nodes[index]]
                If loop = null
                    Break
                Boolean found = false
                Node loop2 = loop
                While loop2 ≠ null
                    If loop2 = start
                        found = true
                        Break
                    loop2 = loopParents[loop2]
                If Not found
                    Break
        newLoopStartToEnd[start] = nodes[index - 1]


List<Node> fixOrder(List<Node> nodes, Map<Node, Node> loopStartToLoopEnd,
Set<LoopInfo> &loops, Set<Node> deadEnds)
    Map<Node, Node> loopParents
    Map<Node, Set<Node>> breakNodesMissed
    Map<Node, Node> loopSuccessors = computeLoopSuccessors(nodes,
loopStartToLoopEnd, deadEnds, loopParents, breakNodesMissed)
    Set<Node> breakNodesMissedSet
    For (node, nodes2) ∈ breakNodesMissed
        breakNodesMissedSet = breakNodesMissedSet ⋃ {nodes2}

    Map<Node, Set<Node>> deadEnds2 = expandDeadEnds(deadEnds)
    Map<Node, Int> deadEndSuccessorsCount
    For (start, succ) ∈ loopSuccessors
        If succ ∈ deadEnds
            deadEndSuccessorsCount[succ] = deadEndSuccessorsCount[succ] + 1

    Set<Node> deadEnds3
    For (node, count) ∈ deadEndSuccessorsCount
```

```
        deadEnds3 = deadEnds3 ⋃ deadEnds2[node]
    deadEnds3 = deadEnds3 - deadEnds

    Stack<Node> loopStartStack
    Stack<Node> loopEndStack
    List<Node> nodes2
    For node ∈ nodes
        If node ∈ breakNodesMissedSet ⋃ deadEnds3 or (node ∈ deadEnds and
deadEndSuccessorsCount[node] > 0)
            Continue
        nodes2 = nodes2 ⋃ {node}
        If loopStartToLoopEnd[node] ≠ null
            Push node onto loopStartStack
            Push loopStartToLoopEnd[node] onto loopEndStack
        While node = Top of loopEndStack
            nodes2 = nodes2 ⋃ breakNodesMissed[Top of loopStartStack]
            Node succ = loopSuccessors[Top of loopStartStack]
            If succ ≠ null
                deadEndSuccessorsCount[succ] = deadEndSuccessorsCount[succ] -
1
            Pop from loopStartStack
            Pop from loopEndStack

    Map<Node, Node> loopStartToEnd2 = newLoopStartToEnd(nodes2, nodes, loops,
loopSuccessors, loopParents, deadEnds2, breakNodesMissed);
    For (start, end) ∈ loopStartToEnd2
        LoopInfo info
        info.start = start
        info.end = end
        info.successor = loopSuccessors[start]
        Add info to loops
    Return nodes2


List<Node> order(Set<Node> nodes, Node start, Set<LoopInfo> &loops)
    Map<Node, Node> loopStartToLoopEnd
    Set<Node> deadEnds
    List<Node> nodes2 = orderBeforeFix(nodes start, loopStartToLoopEnd,
deadEnds, false)
    nodes2 = fixOrder(nodes2, loopStartToLoopEnd, loops, deadEnds)
    Return nodes2
```

Bibliography

[1] Dennis, J. B. A parallel program execution model supporting modular software construction. In Third Working Conference on Massively Parallel Programming Models. IEEE Computer Society, 1998, pp 50-60.

[2] Dennis, J. B. Fresh Breeze: A multiprocessor chip architecture guided by modular programming principles. ACM Sigarch News, March 2003.

[3] Dennis, J. B. The Fresh Breeze Model of Thread Execution. Workshop on Programming Models for Ubiquitous Parallelism, with PACT-2006, Seattle, September 16, 2006.

[4] Emami, M. A Practical Interprocedural Alias Analysis for an Optimizing / Parallelizing Compiler. July 1993.

[5] Ginzburg, I. Compiling Array Computations for the Fresh Breeze Parallel Processor. Thesis for the Master of Engineering degree, MIT Department of Electrical Engineering and Computer Science, May 2007.

[6] S. Skedzielewski, and J. Glauert. IF1: An Intermediate Form for Applicative Languages. Technical Report M-170, Version 1.0, Lawrence Livermore National Laboratory, July 1985.

[7] Wadler, P. Is There a Use for Linear Logic? Partial Evaluation and Semantics-Based Program Manipulation (PEPM), ACM Press, New Haven, Connecticut, June 1991.